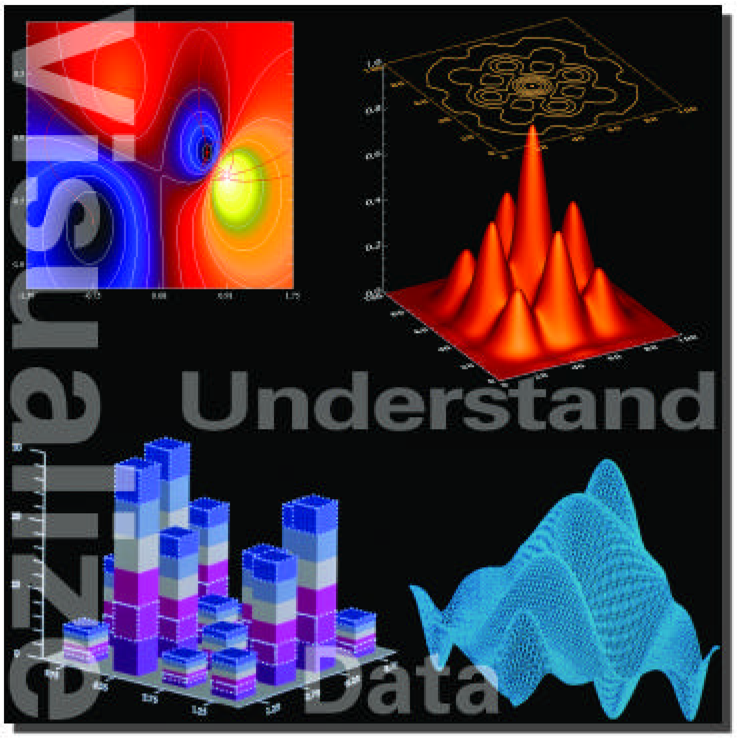
PV-WAVE 7.5



New Features Guide

HELPING CUSTOMERS SOLVE COMPLEX PROBLEMS

Visual Numerics, Inc.

Visual Numerics, Inc. 2500 Wilcrest Drive Suite 200 Houston, Texas 77042-2579 United States of America 713-784-3131 800-222-4675 (FAX) 713-781-9260 http://www.vni.com e-mail: info@boulder.vni.com	Visual Numerics, Inc. (France) S.A.R.L. Tour Europe 33 place des Corolles Cedex 07 92049 PARIS LA DEFENSE FRANCE +33-1-46-93-94-20 (FAX) +33-1-46-93-94-39 e-mail: info@vni-paris.fr	Visual Numerics International, Ltd. Suite 1 Centennial Court East Hampstead Road Bracknell, Berkshire RG 12 1 YQ UNITED KINGDOM +01-344-458-700 (FAX) +01-344-458-748 e-mail: info@vniuk.co.uk
Visual Numerics, Inc. 7/F, #510, Sect. 5 Chung Hsiao E. Rd. Taipei, Taiwan 110 ROC +886-2-727-2255 (FAX) +886-2-727-6798 e-mail: info@vni.com.tw	Visual Numerics International GmbH Zettachring 10 D-70567 Stuttgart GERMANY +49-711-13287-0 (FAX) +49-711-13287-99 e-mail: info@visual-numerics.de	Visual Numerics Japan, Inc. Gobancho Hikari Building, 4th Floor 14 Gobancho Chiyoda-Ku, Tokyo, 102 JAPAN +81-3-5211-7760 (FAX) +81-3-5211-7769 e-mail: vda-sprt@vnij.co.jp
VIsual Numerics S.A. de C.V. Cerrada de Berna 3, Tercer Piso Col. Juarez Mexico, D.F. C.P. 06600 Mexico	Visual Numerics, Inc., Korea Rm. 801, Hanshin Bldg. 136-1, Mapo-dong, Mapo-gu Seoul 121-050 Korea	

© 1990-2001 by Visual Numerics, Inc. An unpublished work. All rights reserved. Printed in the USA. 2001

Information contained in this documentation is subject to change without notice.

IMSL, PV- WAVE, Visual Numerics and PV-WAVE Advantage are either trademarks or registered trademark s of Visual Numerics, Inc. in the United States and other countries.

The following are trademarks or registered trademarks of their respective owners: Microsoft, Windows, Windows 95, Windows NT, Fortran PowerStation, Excel, Microsoft Access, FoxPro, Visual C, Visual C++ — Microsoft Corporation; Motif — The Open Systems Foundation, Inc.; PostScript — Adobe Systems, Inc.; UNIX — X/Open Company, Limited; X Window System, X11 — Massachusetts Institute of Technology; RISC System/6000 and IBM — International Business Machines Corporation; Java, Sun — Sun Microsystems, Inc.; HPGL and PCL — Hewlett Packard Corporation; DEC, VAX, VMS, OpenVMS — Compaq Computer Corporation; Tektronix 4510 Rasterizer — Tektronix, Inc.; IRIX, TIFF — Silicon Graphics, Inc.; ORACLE — Oracle Corporation; SPARCstation — SPARC International, licensed exclusively to Sun Microsystems, Inc.; SYBASE — Sybase, Inc.; HyperHelp — Bristol Technology, Inc.; dBase — Borland International, Inc.; MIFF — E.I. du Pont de Nemours and Company; JPEG — Independent JPEG Group; PNG — Aladdin Enterprises; XWD — X Consortium. Other product names and companies mentioned herein may be the trademarks of their respective owners.

IMPORTANT NOTICE: **Use of this document is subject to the terms and conditions of a Visual Numerics Software License Agreement, including, without limitation, the Limited Warranty and Limitation of Liability.** If you do not accept the terms of the license agreement, you may not use this documentation and should promptly return the product for a full refund. Do not make illegal copies of this documentation. No part of this documentation may be stored in a retrieval system, reproduced or transmitted in any form or by any means without the express written consent of Visual Numerics, unless expressly permitted by applicable law.

The Visualizaton Toolkit

Copyright (c) 1993-1995 Ken Martin, Will Schroeder, Bill Lorensen.

This software is copyrighted by Ken Martin, Will Schroeder and Bill Lorensen. The following terms apply to all files associated with the software unless explicitly disclaimed in individual files. This copyright specifically does not apply to the related textbook "The Visualization Toolkit" ISBN 013199837-4 published by Prentice Hall which is covered by its own copyright.

The authors hereby grant permission to use, copy, and distribute this software and its documentation for any purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. Additionally, the authors grant permission to modify this software and its documentation for any purpose, provided that such modifications are not distributed without the explicit consent of the authors and that existing copyright notices are retained in all copies. Some of the algorithms implemented by this software are patented, observe all applicable patent law.

IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Table of Contents

Preface

This *New Features Guide* provides detailed information on major features added to PV-WAVE, PV-WAVE:IMSL Mathematics, and PV-WAVE:IMSL Statistics since version 7.0.

For additional information on this release, see the Release Notes, available in the PV-WAVE installation directory or on our Web site: `www.vni.com.`

What's in this Manual

Chapter 1: ***New Features Introduction*** — Provides an overview of the scope of the PV-WAVE 7.5 release, along with user information for major enhancements. added to PV-WAVE since version 7.0. The information in this chapter is integrated into the *PV-WAVE Programmer's Guide, User's Guide, and Application Developer's Guide*, the *PV-WAVE:IMSL Statistics Reference*, and the *PV-WAVE:IMSL Mathematics Reference.*

Chapter 2: ***New Commands*** — An alphabetical listing and detailed description of the new PV-WAVE procedures and functions. The information in this chapter is integrated into the *PV-WAVE Reference*, *PV-WAVE Database Connection User's Guide*, *PV-WAVE:IMSL Statistics Reference*, and *PV-WAVE:IMSL Mathematics Reference*.

Chapter 3: ***Updates to Existing Functionality*** — Describes additions to previously existing functionality. This chapter already is integrated into the *PV-WAVE Reference*, *PV-WAVE:IMSL Statistics Reference*, and *PV-WAVE:IMSL Mathematics Reference*.

Index — Contains page references for an alphabetical list of subjects described in this guide.

Conventions Used in this Manual

You will find the following conventions used throughout this manual:

- Code examples appear `in this typeface`. For example:

```
PLOT, temp, s02, Title = 'Air Quality'
```

- Code comments are shown in this typeface, immediately below the commands they describe. For example:

```
PLOT, temp, s02, Title = 'Air Quality'
   ; This command plots air temperature data vs. sulphur
   ; dioxide concentration.
```

- Variables are shown in lowercase italics (*myvar*), function and procedure names are shown in uppercase (XYOUTS), keywords are shown in mixed case italic (*XTitle*), and system variables are shown in regular mixed case type (!Version). For better readability, all GUI development routines are shown in mixed case (WwMainMenu).

- A `$` at the end of a line of PV-WAVE code indicates that the current statement is continued on the following line. By convention, use of the continuation character (`$`) in this document reflects its syntactically correct use in PV-WAVE. This means, for instance, that *strings* are never split onto two lines without the addition of the string concatenation operator (+). For example, the following lines would produce an error if entered literally in PV-WAVE.

```
WAVE> PLOT, x, y, Title = 'Average $
   Air Temperatures by Two-Hour Periods'
      ; Note that the string is split onto two lines; an error
      ; message is displayed if you enter a string this way.
```

  The correct way to enter these lines is:

```
WAVE> PLOT, x, y, Title = 'Average ' + $
   'Air Temperatures by Two-Hour Periods'
      ; This is the correct way to split a string onto two
      ; command lines.
```

- Reserved words, such as FOR, IF, CASE, are always shown in uppercase.

Technical Support

If you have problems installing, unlocking, or running your software, contact Visual Numerics Technical Support by calling:

Office Location	**Phone Number**
Corporate Headquarters Houston, Texas	713-784-3131
Boulder, Colorado	303-939-8920
France	+33-1-46-93-94-20
Germany	+49-711-13287-0
Japan	+81-3-5211-7760
Korea	+82-2-3273-2633
Mexico	+52-5-514-9730
Taiwan	+886-2-727-2255
United Kingdom	+44-1-344-458-700

Users outside the U.S., France, Germany, Japan, Korea, Mexico, Taiwan, and the U.K. can contact their local agents.

Please be prepared to provide the following information when you call for consultation during Visual Numerics business hours:

- Your license number, a six-digit number that can be found on the packing slip accompanying this order. (If you are evaluating the software, just mention that you are from an evaluation site.)
- The name and version number of the product. For example, PV-WAVE 7.5.
- The type of system on which the software is being run. For example, SPARCstation, IBM RS/6000, HP 9000 Series 700.
- The operating system and version number. For example, HP-UX 10.2 or IRIX 6.5.
- A detailed description of the problem.

FAX and E-mail Inquiries

Contact Visual Numerics Technical Support staff by sending a FAX to:

Office Location	**FAX Number**
Corporate Headquarters	713-781-9260
Boulder, Colorado	303-245-5301
France	+33-1-46-93-94-39
Germany	+49-711-13287-99
Japan	+81-3-5211-7769
Korea	+82-2-3273-2634
Mexico	+52-5-514-5880
Taiwan	+886-2-2727-6798
United Kingdom	+44-1-344-458-748

or by sending E-mail to:

Office Location	**E-mail Address**
Boulder, Colorado	`support@boulder.vni.com`
France	`support@vni-paris.fr`
Germany	`support@visual-numerics.de`
Japan	`vda-sprt@vnij.co.jp`
Korea	`support@vni.co.kr`
Taiwan	`support@vni.com.tw`
United Kingdom	`support@vniuk.co.uk`

Electronic Services

Service	Address
General e-mail	`info@boulder.vni.com`
Support e-mail	`support@boulder.vni.com`
World Wide Web	`http://www.vni.com`
Anonymous FTP	`ftp.boulder.vni.com`
FTP Using URL	`ftp://ftp.boulder.vni.com/VNI/`
PV-WAVE Mailing List:	`Majordomo@boulder.vni.com`
To subscribe include:	`subscribe pv-wave YourEmailAddress`
To post messages	`pv-wave@boulder.vni.com`

1

CHAPTER

New Features Introduction

This chapter provides an introduction to the scope of the PV-WAVE 7.5 release, along with user information for major enhancements added to PV-WAVE since version 7.0.

For information on specific new PV-WAVE functions and procedures, see *New PV-WAVE Commands* in Chapter 2, *New Commands*.

For information on updates to existing PV-WAVE functionality, see Chapter 3, *Updates to Existing Functionality*.

PV-WAVE 7.5 Major Enhancements

PV-WAVE VTK Integration

PV-WAVE has been updated to include the integration of the Visualization Toolkit (VTK) to make all of its functionality available to PV-WAVE users. PV-WAVE users can now create high quality, interactive graphics through the use of the PV-WAVE link to the Visualization Toolkit (VTK). The Visualization Toolkit is an Open Source toolkit for creating both simple and complex visualizations in 3D using OpenGL for high-performance, accelerated graphics. The Visualization Toolkit and PV-WAVE complement each other well. PV-WAVE excels at data access, data manipulation, numerical algorithms, data filtering, user-interface development, and many interactive 2D graphical tasks. The Visualization Toolkit is a best-of-breed tool for creating complex 3D visualizations. Together they provide a simple and quick way to build tools for Visual Data Analysis. For detailed information on the new PV-WAVE-VTK functions, see *New PV-WAVE Commands* in Chapter 2, *New Commands*.

PV-WAVE: IMSL Mathematics and Statistics

PV-WAVE: IMSL Mathematics and PV-WAVE: IMSL Statistics have been updated to include a complete implementation of Visual Numeric's IMSL C Numerical Library 5.0.

New Functionality includes:

- A new time series routine useful in navigation, surveying, vehicle tracking (aircraft, spacecraft, missiles), geology, oceanography, fluid dynamics, steel/paper/power industries, and demographic estimation.
- 50 new functions in the area of finance and bonds.
- Routines to compute low discrepancy series of points using a generalized Faure sequence.
- A new algorithm for efficient multi-dimensional quadrature.
- More than twenty new random number routines, including Generalized Feedback Shift Register (GFSR) generator support.

This integration represents a major enhancement to the WAVE Family. For detailed descriptions of the new mathematics and statistics routines, see *New PV-WAVE:IMSL Mathematics Commands* and *New PV-WAVE:IMSL Statistics Commands* in Chapter 2, *New Commands*. For of the new mathematics and statistics functionality, see the PV-WAVE*:IMSL Statistics Reference* and the PV-WAVE*:IMSL Mathematics Reference*.

PV-WAVE: Database Connection

New functionality includes:

- **DB_GET_BINARY Function** - Returns binary large objects (BLOBS) from a DBMS (database management system) server.
- **NULL_PROCESSOR Function** - Facilitates the use of the resulting output from the *Null_Info* keyword for the DB_SQL function by extracting locations of NULL values in the database query.
- **Keyword - null_info_object** to DB_SQL function, which provides information on the location of NULL data in the result from a database query. For more information on keyword null_info_object, see *NULL_PROCESSOR Function* on page 92.

For more detailed information on the new database connection functions, see *New PV-WAVE:Database Connection Functions* on page 91.

PV-WAVE: DICOM Reader

PV-WAVE users can now read and write DICOM images with the implementation of IMAGE_READ and IMAGE_WRITE functions. For more details on IMAGE_READ and IMAGE_WRITE, see the *PV-WAVE Reference*.

New Functionality includes:

DICM_TAG_INFO Function — extracts the tag information from an image associative array that contains a DICOM image. For more details on DICM_TAG_INFO, see *New PV-WAVE Commands* on page 5 in Chapter 2, *New Commands*.

2

CHAPTER

New Commands

This chapter lists and describes new functions and procedures added to:

- PV-WAVE (page 5)
- PV-WAVE:Database Connection (page 91)
- PV-WAVE:IMSL Mathematics (see page 95)
- PV-WAVE:IMSL Statistics (page 190)

New PV-WAVE Commands

This section describes the new functions and procedures that have been added to PV-WAVE 7.5.

AFFINE Function

Standard Library function that applies an affine transformation to an array.

Usage

result = AFFINE(*a*, *b*, [*c*])

Input Parameters

a — An n-dimensional array.

b — An invertible (n, n) array.

c — An n-element vector (optional).

Returned Value

result — An array representing *a* (and of the same dimensions as *a*) under the coordinate transformation `y = b#x + c`, where `y` and `x` are coordinates for the *result* and for *a*, respectively, which differ from array index coordinates by a simple translation to the array centroid.

Keywords

None.

Example

See `wave/lib/user/examples/affine_ex`.

See Also

ROT, ROTATE, ROT_INT

BLOB Function

Standard Library function that isolates a homogeneous region in an array.

Usage

result = BLOB(*a*, *i*, *b*)

Input Parameters

a — An array of n dimensions.

i — A vector of n integers giving a seed element for the region.

b — A two-element vector giving bounds for values in the region.

Returned Value

result — An (m,n) array of m n-dimensional indices into *a*. *result* defines the region containing *i* whose values lie in the range [b(0),b(1)]. If no such region exists then result is returned as -1.

Keywords

k — A positive integer (less than or equal to *n*) controlling connectivity: two cells are connected if they share a common boundary point, and if their centroids are within the square root of *k* of each other. *k* = 1 by default, which implies connected cells share a common face.

Example

See `wave/lib/user/examples/blob_grow.pro`

See Also

BLOBCOUNT, BOUNDARY, NEIGHBORS

BLOBCOUNT Function

Standard Library function that counts homogeneous regions in an array.

Usage

result = BLOBCOUNT(*a*, *b*)

Input Parameters

a — An array of *n* dimensions.

b — A two-element vector of bounds for values in a region.

Returned Value

result — A list in which each element defines a distinct region whose values lie in the range [b(0),b(1)]. *result*(j) is a (m(j),n) array of m(j) n-dimensional indices into *a*. If no such regions exist, then *result* is returned as -1.

Keywords

k — A positive integer (less than or equal to *n*) controlling connectivity: two cells are connected if they share a common boundary point, and if their centroids are within the square root of *k* of each other. *k* = 1 by default, which implies that connected cells share a common face.

Example 1

```
a = ( image_read(!data_dir+'vni_small.tif') )( 'pixels' )
a = bytscl( resamp(a,500,500) )  &  tv, a
r = blobcount( a, [255,255] )
for i = 0, n_elements(r)-1 do a(index_conv(a,r(i))) = 50  &  tv, a
```

Example 2

See `wave/lib/user/examples/blobcount_ex.pro`

See Also

BLOB, BOUNDARY, NEIGHBORS

BOUNDARY Function

Standard Library function that computes the boundary of a region in an array.

Usage

result = BOUNDARY(*a*,*r*)

Input Parameters

a — An array of *n* dimensions.

r — A vector of indices defining the region of *a*.

Returned Value

result — A vector of indices defining the boundary of *r*.

Keywords

k — A positive integer (less than or equal to *n*) defining connectivity. A boundary element of *r* is an element of *r* with neighbors not in *r*; two array cells are neighbors if they share a common boundary point and their centroids are within the square root of *k* of each other. *k* = 1 by default, which implies neighbors share a common face.

Examples

```
a = indgen( 5, 4 )  &  pm, a
print, fix(boundary(a,[1,2,3,6,7,8,12,13]))
print, fix(boundary(a,[1,2,3,6,7,8,12,13],k=2))
a = bytscl( dist(500) )  &  r = where( 150 le a and a le 200 )
a( boundary(a,r) ) = 0   &  tv, a
```

See Also

BLOB, BLOBCOUNT, NEIGHBORS

CPROD Function

Standard Library function that returns the Cartesian product of some arrays.

Usage

result = CPROD(*a*)

Input Parameters

a — A list of *n* arrays.

Returned Value

result — An (m,n) array where *result(i,*)* is an element of the Cartesian product of the n arrays in *a*, and where *result(*,j)* contains only elements from *a(j)*; *result* is ordered so that *result(*,j)* cycles through the elements of *a(j)* in order, and does so faster than *result(*,j+1)* cycles through the elements of *a(j+1)*.

Keywords

None.

Example

```
pm, cprod( list( [0,1], [0,1,2], [0,1,2,3] ) )
```

CURVATURES Function

Standard Library function that computes curvatures on a parametrically defined surface.

Usage

c = curvatures (*s*)

Input Parameters

s — A 3-element list of 2-dimensional arrays of dimensions *d*.

Returned Value

c — A 2-element list of 2-dimensional arrays of dimensions *d*, where *c*(0) defines the distribution of minimum curvature and *c*(1) defines the distribution of maximum curvature.

Keyword

x **-** A 2-element list of vectors defining the independent variables. By default, *x*(*i*) = findgen(*d*(*i*))

Example

See `wave/lib/user/examples/curvatures_ex.pro`.

See Also

EUCLIDEAN, JACOBIAN, NORMALS

DERIVN Function

Standard Library function that differentiates a function represented by an array.

Usage

result = DERIVN(*a*, *n*)

Input Parameters

a — An array of values of the dependent variable.

n — An integer (≥ 0) designating which dimension to differentiate.

Returned Value

result — An array of the same dimensions as *a*, representing the derivative with respect to the *n*th independent variable.

Keywords

x — A vector defining the independent variable of differentiation. *x* defaults to the indices into dimension *n* of *a*.

Examples

```
pm, derivn( [0,2,1,0,1], 0 )
pm, derivn( [[0,2,1,0,1],[2,1,0,2,0],[1,0,2,1,2]], 0 )
pm, derivn( [[0,2,1,0,1],[2,1,0,2,0],[1,0,2,1,2]], 1 )
```

See Also

DERIV

DICM_TAG_INFO Function

Extracts Digital Imaging and Communications in Medicine (DICOM) tags information from an image associative array.

Usage

result = DICM_TAG_INFO (filename, image)

Input Parameters

filename **—** On input, a string containing the name of the file which contains the descriptions for the DICOM tags.

image**—** An associative array in image format.

Returned Value

result **—** An associative array containing DICOM tags information.

Discussion

The DICM_TAG_INFO function extracts the tag information from an image associative array that contains a DICOM image. The tag information is returned as an associative array. The following table describes each key of the associative array:

Array Key Name	**Variable Type**	**Description**
tag	STRING	A 1-dimensional array containing the DICOM tags
description	STRING	A 1-dimensional array containing the DICOM tag descriptions
value	STRING	A 1-dimensional array containing the DICOMtag values

The DICM_TAG_INFO function needs a file containing the tag description as input. This file contains a tag followed by a description for this tag. The tags in this file must be in ascending order. For example:

```
(0002,0000) Group Length UL 1
(0002,0001) File Meta Information Version OB 1
```

Example

This example uses IMAGE_READ to read a DICOM image file. Then it extracts the DICOM tags and displays the information of the result variable.

```
image = IMAGE_READ('test.dicm', File_type='dicm')
tags = DICM_TAG_INFO('dict.txt',image)
INFO, tags, /Full

TAGS            AS. ARR   = Associative Array(3)
  tag             STRING    = Array(36)
  description     STRING    = Array(36)
  value           STRING    = Array(36)
```

See Also

IMAGE_READ, IMAGE_WRITE

EUCLIDEAN Function

Standard Library function that transforms the Euclidean metric for a Jacobian *j* = Jacobian (*f*)

Usage

e = euclidean (*j*)

Input Parameters

j — A Jacobian defined by a *n*-element list of *m*-element lists of *m*-dimensional arrays of dimensions *d*.

Returned Value

e — The Euclidean metric under a transformation with Jacobian *j*: an *m*-element list of *m*-element lists of *m*-dimensional arrays. (*e*(*p*))(*q*) is the *m*-dimensional array (of dimensions *d*) that represents the (*p*, *q*) component of the metric.

Keywords

None.

Example

See `wave/lib/user/examples/euclidean_ex.pro`.

See Also

CURVATURES, JACOBIAN, NORMALS

EXPAND Function

Standard Library function that expands an array into higher dimensions.

Usage

result = EXPAND(*a, d, i*)

Input Parameters

a — An array of *n* dimensions.

d— A vector specifying the dimensions for the new array.

i — A monotonically increasing vector of n indices into *d* specifying which of the new dimensions correspond to old dimensions:
d(i) must equal `SIZE(`*a*, `/Dimensions`).

Returned Value

result — An array of dimensions *d*, the expansion of the input array.

Keywords

None.

Examples

```
pm, EXPAND( [0,1], [2,3], [0] )
```

```
pm, EXPAND( [0,1], [3,2], [1] )
pm, EXPAND( [[0,1,2],[3,4,5]], [5,3,2], [1,2] )
pm, EXPAND( [[0,1,2],[3,4,5]], [3,5,2], [0,2] )
pm, EXPAND( [[0,1,2],[3,4,5]], [3,2,5], [0,1] )
```

See Also

REBIN, REPLV

EXPON Function

Standard Library function that performs general exponentiation.

Usage

result = EXPON(*a*, *b*)

Input Parameters

a — An array (scalar) of any numerical data type.

b — An array (scalar) of any numerical data type.

Returned Value

result — A double complex array (scalar) containing the values a^b.

Keywords

None.

Examples

```
pm, EXPON( [complex(0,1),-1], [complex(2,3),0.5] )
```

EXTREMA Function

Standard Library function that finds the local extrema in an array.

Usage

result = EXTREMA(*array*)

Input Parameters

array — The array for which the local extrema will be found.

Returned Value

result — A list containing two vectors of indices into *array*. *result*(0) contains the local minima and *result*(1) contains the local maxima

Keywords

None.

Examples

```
e = EXTREMA( [0,1,2,2,2,3,2,1,3] )   &  pm, e(0), ’ ’  &  pm, e(1)
a = bytscl( randomu(s,5,5), top=9 )  &  pm, a
e = EXTREMA( a )                     &  pm, e(0), ’ ’  &  pm, e(1)
```

See Also

MAX, MIN

FACTOR Function

Standard Library function that returns the prime factorization of an integer greater than 1.

Usage

result = FACTOR(*i*)

Input Parameters

i — An integer greater than 1.

Returned Value

result — Sorted vector of longs containing the prime factorization of *i*.

Keywords

a — If set, *result* contains all factors instead of just prime factors.

Examples

```
pm, FACTOR(12,/a)
pm, FACTOR(12)
```

See Also

GCD, LCM, PRIME

GCD Function

Standard Library function that returns the greatest common divisor of some integers greater than 0.

Usage

result = GCD(*i*)

Input Parameters

i — An array of integers greater than 0.

Returned Value

result — An integer: the greatest common divisor of the integers *i*.

Keywords

None.

Examples

```
pm, GCD( [12,20,32] )
```

See Also

FACTOR, LCM, PRIME

GREAT_INT Function

Greatest Integer Function. Standard Library function that returns the greatest integer less than or equal to the passed value. Also known as the Floor Function.

Usage

result = GREAT_INT(*values*)

Input Parameters

values — An array (scalar).

Returned Value

result — A long array (scalar) of the same dimensions as *values*: *result*(i) is the greatest integer less than or equal to *values*(i).

Keywords

None.

Examples

```
PM, GREAT_INT( [-0.5,0,0.5] )
```

See Also

SMALL_INT

GRIDN Function

Standard Library function that grids n dimensional data.

Usage

result = GRIDN(*d*, *i*)

Input Parameters

d — An (m,n+1) array of m datapoints in n independent variables and one dependent variable; *d(*,n)* is the dependent variable.

i — A vector of n integers specifying the dimensions of the grid.

Returned Value

result — An n dimensional array of values of the dependent variable on a regular grid over the independent variables.

Keywords

r — A scalar specifying the order of the weighting function. The dependent variable at a grid point is computed as a weighted average of the variable over all neighborhood datapoints. The weighting function is $1/e^r$ where e is the Euclidean distance between the grid point and the datapoint. *r* defaults to 2

t — A scalar between 0 and 1 specifying neighborhood size. *t=1* gives a maximal neighborhood which includes all datapoints, while lower *t* values yield smaller neighborhoods. *t* defaults to 1

b — A 2 x n array fixing the boundary of the grid. *b(0,*)* is the minimum corner and *b(1,*)* is the maximum corner. The default extent of the grid is the same as that of the data.

f — The name of a user-supplied procedure describing voids in the independent variable space (datapoints and gridpoints within these regions are ignored in computation). Input to *f* is a (p,n) array of p points in the independent variable space. *f* outputs two items where the first item is a vector of indices indicating which of the p input points are within bounds, and where the second item is a scalar that will appear as a place holder for the dependent variable at out-of-bounds gridpoints.

c — (output) A list of n vectors defining the grid coordinates.

Examples

See `wave/lib/user/examples/gridnex1.pro`

`wave/lib/user/examples/gridnex2.pro`

See Also

FAST_GRID2, FAST_GRID3, FAST_GRID4, INTERPOLATE

HISTN Function

Standard Library function that computes an n dimensional histogram.

Usage

result = HISTN(*d* [, *axes*])

Input Parameters

d — An (m,n) array of m data points in n-space.

Returned Value

result — An n dimensional array of size *binnum*.

axes = (optional - use with */scale*) an (n, *binnum*) array containing properly scaled axes with which to plot the results. For example:

```
CONTOUR, result, transpose(axes(0,*)), transpose(axes(1,*))
```

Keywords

Binnum — The number of bins for the histogram.

Binsize — The size of bins for the histogram. The default = 1.

NOTE Only 1 of *binnum* or *binsize* can be set.

/Scale — If set, the result is scaled so to have unit volume under the curve/surface when plotted against x.i/stdev(x.i)

/Compatible — If set, the result will align with HISTOGRAM. The default behavior of HISTN is "binnum-central" logic, while the default behavior of

HISTOGRAM is “binsize-central” logic. Setting */compatible* will force HISTN to be “binsize-central.”

NOTE When using */compatible* with 2D arrays and setting *binsize* manually, you may see poor results if the *binsize* is not appropriate for all variables. In this case, you should either set *binnum* or not use */compatible*.

Examples

Interpreting an n-dimensional histogram, a 2D example.

Consider two sets of 10 random numbers. If one computes 1D histograms with 3 bins, you may find these results (number of items in each bin):

```
x = RANDOMN(s,10)   &  y = RANDOMN(s,10)

xy = FLTARR(10,2)   &  xy(*,0) = x     &  xy(*,1) = y

PRINT, HISTN(x, binnum=3)

           4           5           1

PRINT, HISTN(y, binnum=3)

           3           4           4
```

The result for the 2D histogram may give:

```
PRINT, HISTN(xy, binnum=3)

           1           1           1

           2           2           0

           1           2           0
```

Summing this result vertically yields the 1D result for *x*. Summing this result horizontally yields the 1D result for *y*. To plot a probability distribution function with unit volume under the surface, one would call HISTN with the /Scaled keyword and plot the results against x/std(x) and y/STD(y). Using the *axes* output parameter with HISTN returns these vectors in a 2D array.

```
h2 = HISTN(xy, axes, binnum=3, /scaled)

shade_surf, h2, axes(0,*), axes(1,*)
```

See Also

HISTOGRAM

INDEX_AND Function

Standard Library function that computes the logical AND for two vectors of positive integers.

Usage

result = INDEX_AND(*array*$_1$, *array*$_2$)

Input Parameters

***array*$_1$** — A vector of positive integers.

***array*$_2$** — A vector of positive integers.

Returned Value

result — A vector containing all elements common to both input arrays. (*result* is not unique'd.)

Keywords

None.

Examples

```
PM INDEX_AND( [2,0,3], [1,2,0,2] )
```

See Also

INDEX_OR, WHEREIN

INDEX_OR Function

Standard Library function that computes the logical OR for two vectors of positive integers.

Usage

result = INDEX_OR(*array*$_1$, *array*$_2$)

Input Parameters

array$_1$ — A vector of positive integers.

array$_2$ — A vector of positive integers.

Returned Value

result — A vector consisting of all elements contained in either of the input arrays (*result* is unique'd).

Keywords

None.

Examples

```
PM, INDEX_OR( [2,0,3], [1,2,0,2] )
```

See Also

INDEX_AND, WHEREIN

INTERPOLATE Function

Standard Library function that interpolates scattered data at scattered locations.

Usage

result = INTERPOLATE(*d*, *x*)

Input Parameters

d — An (m, n+1) array of m datapoints in n independent variables and one dependent variable; *d(*,n)* is the dependent variable.

x — A (p,n) array specifying p interpolation points.

Returned Value

result — A 1d array of values of the dependent variable at points ***x***.

Keywords

r — A scalar specifying the order of the weighting function. The dependent variable at an interpolation point is computed as a weighted average of the variable over all datapoints. The weighting function is $1/e^r$ where *e* is the Euclidean distance between the interpolation point and the datapoint. *r* defaults to 2

Examples

```
x = findgen(51) / 50    &   plot, x, INTERPOLATE([[0,1],[2,3]],x)
```

See Also

GRIDN

INTRP Function

Standard Library function that interpolates an array along one of its dimensions.

Usage

result = INTRP(*a*, *n*, *x*)

Input Parameters

a — An array.

n — An integer (≥0) designating the dimension to interpolate.

x — A one-dimensional array giving the coordinates at which to interpolate.

Returned Value

result — The array of interpolated slices perpendicular to dimension n.

Keywords

z — a strictly increasing 1d array of coordinates for dimension n (defaults to the indicies into this dimension).

Examples

See `wave/lib/user/examples/intrp_ex.`

See Also

REBIN, RESAMP

JACOBIAN Function

Standard Library function that computes the Jacobian of a function represented by *n* *m*-dimensional arrays

Usage

j = jacobian (*f*)

Input Parameters

f — an *n*-element list of *m*-dimensional arrays all of the same dimensions *d*; *f* represents a *n*-valued function of *m* variables.

Returned Value

j — an *n*-element list of *m*-element lists of *m*-dimensional arrays: (*j*(*p*))(*q*) is the *m*-dimensional array (of dimensions *d*) which represents the derivitive of the *p*th dependent variable with respect to the *q*th independent variable.

Keywords

x — *m*-element list of vectors defining the independent variables; by default, *x*(*i*) = findgen(*d*(*i*)).

Example

```
f = list( randomu(s,10,20), randomu(s,10,20), randomu(s,10,20) )
j = jacobian( f )
for p=0,2 do for q=0,1 do pm, same( (j(p))(q), derivn(f(p),q) )
```

See Also

CURVATURES, DERIVN, EUCLIDEAN, NORMALS

LCM Function

Standard Library function that returns the least common multiple of some integers greater than 1.

Usage

result = LCM(*i*)

Input Parameters

i — An array of integers greater than 1.

Returned Value

result — An integer, the least common multiple of the integers i.

Keywords

None.

Examples

```
pm, LCM( [3,2,4] )
```

See Also

FACTOR, GCD, PRIME

LISTARR Function

Returns a list.

Usage

result = LISTARR(*number_elements,[value]*)

Input Parameters

number_elements — The number of elements the created list should contain. Must be a scalar expression.

value — (optional) The value with which to initialize each element of the list. May be any data type.

Returned Value

result — A list with the requested number of elements.

Keywords

None.

Discussion

Values in the list are initialized to 0L unless *value* is specified.

Examples

```
INFO, LISTARR( 2, 10 ), /Full
```

MINIMIZE Function

Standard Library function that minimizes a real valued function of n real variables.

Usage

x = MINIMIZE(*f, l, u, g, i, y*)

Input Parameters

f — A string specifying a user supplied function to be minimized. Input is a (m,n) array of m points in n-space (m variable); output is a (m,p+1) array b, where p is the number of constraints, b(*,0) contains the objective function values at each of the m input points, and b(*,j) contains the corresponding values of the j'th constraint. All constraints must be of the form c(x) ≤ 0.

l — n-element vector of lower bounds for the independent variables

u — n-element vector of upper bounds for the independent variables

g — n-element vector giving an initial guess for the solution

i — An integer limit on the number of iterations

Returned Value

x — The n-element solution vector

y — (optional) A (p+1)-element vector containing the objective function value at x followed by the constraint values at x.

Keywords

d — A string specifying a user-supplied gradient function. Input is the n-element vector at which to calculate the gradient(s). Output is a (n,p+1) array that contains the objective function gradient followed by the constraint gradients.

s — An (n,2) array where s(*,0) is the maximum allowable step and s(*,1) is the minimum allowable step. The default is [[(u-l)/100], [(u-l)/1000]].

Examples

See `wave/lib/user/examples/minimize_ex*.pro`.

MOLEC Function

Standard Library function that creates an image of a ball and stick molecular model.

Usage

result = MOLEC(*filename*)

Input Parameters

filename — The name of an ASCII file describing the molecular model. Line 1 in the file consists of a single integer designating the number (m) of atoms. Each of the lines 2-(m+1) contains 7 floats describing an atom in terms of centroid, normalized RGB color components, and diameter. Line m+2 consists of a single integer equal to the number (n) of bonds; each of the lines (m+3)-(m+n+2) contains 6 floats describing a bond as endpoint1 followed by endpoint2.

Returned Value

result — A 24-bit image of the molecular model.

Keywords

h — A scale factor for adjusting atom size. The default is `h=1.0`.

s — A 2-element vector specifying image size. The default is `s=[500,500]`.

v — A 3-by-4 double-precision floating-point array used to override the autogeneration of the view to that specified as: [viewpoint, top_left_viewplane, bottom_left_viewplane, bottom_right_viewplane]. *v* and !P.T control the 3d view.

k — (output) A 3-by-4 double-precision floating-point array used to return the automatically calculated view as: [viewpoint, top_left_viewplane, bottom_left_viewplane, bottom_right_viewplane].

Examples

```
T3D, /Reset  &  TV, MOLEC(!data_dir+'molec.dat',h=0.6), true=3
```

MOMENT Function

Standard Library function that computes moments of an array.

Usage

result = MOMENT(*a*, *i*)

Input Parameters

a — An array of n dimensions.

i — A vector of n non-negative real numbers defining moment order.

Returned Value

result — A scalar double equal to:

$$\sum_{J_{n-1}} \cdots \sum_{J_0} a\left(J_0 \ldots J_{n-1}\right) J_0^{\,i(0)} \ldots J_{n-1}^{\,i(n-1)}$$

Keywords

None.

Examples

```
a = BYTARR( 600, 500 )
x = INDEX_CONV( a, LINDGEN(N_ELEMENTS(a)) )
a( WHERE( (x(*,0)-200)^2+(x(*,1)-300)^2 LT 100^2 ) ) = 255
TV, a
CENTROID = [MOMENT(a,[1,0]),MOMENT(a,[0,1])] / MOMENT(a,[0,0])
PLOTS, CENTROID, /DEVICE, COLOR=0, PSYM=2
```

NEIGHBORS Function

Standard Library function that finds the neighbors of specified array elements.

Usage

result = NEIGHBORS(*a*, *i*)

Input Parameters

a — An array of n dimensions.

i — An m-element vector of m one-dimensional indices into *a*.

Returned Value

result — An (m,*) array of one-dimensional indices into *a*: *result*(j,*) contains i(j) and its neighbors.

Keywords

k — A positive integer (less than or equal to *n*) defining connectivity. Two array cells are neighbors if they share a common boundary point, and if their centroids are within √k of each other. The default is k = 1, which implies neighbors share a common face.

Examples

```
a = INDGEN( 8, 9 )  &  pm, a
pm, fix(NEIGHBORS(a,[0,4,26,47,71]))
pm, fix(NEIGHBORS(a,[0,4,26,47,71],k=2))
```

NORMALS Function

Standard Library function that computes unit normals on a parametrically defined surface.

Usage

n = normals (*j*)

Input Parameters

j — The Jacobian (computed by the JACOBIAN function) on the surface.

Returned Value

n — A 3-element list of 2-dimensional arrays of the same size as those in *j*: *n*(*i*) is the array describing the distribution of the i^{th} component of the unit normal.

Keywords

None.

Example

See `wave/lib/user/examples/normals_ex.pro`.

See Also

CURVATURES, EUCLIDEAN, JACOBIAN

PRIME Function

Standard Library function that returns all positive primes less than or equal to a scalar input.

Usage

result = PRIME(*value*)

Input Parameters

value — The scalar input.

Returned Value

result — A vector containing all positive primes less than or equal to *value*.

Keywords

None.

Examples

```
PRINT, PRIME(10)
```

See Also

FACTOR, GCD, LCM

PRODUCT Function

Returns the product of all elements in an array.

Usage

result = PRODUCT(*array*)

Input Parameters

array — An array.

Returned Value

result — A scalar value equal to the product of all the elements in *array*.

Keywords

None.

Discussion

If *array* is of type single- or double-precision floating point, or single- or double-precision complex, the result will be of the same type. If *array* is of any other type, PRODUCT returns a single-precision floating-point result.

Examples

```
PM, PRODUCT( [2,3,4] )
```

RENDER24 Function

Standard Library function that generates a ray-traced rendered 24-bit image of m objects.

Usage

result = RENDER24(*b*)

Input Parameters

b — A m-element list containing the m objects to render; objects are created using the CONE, CYLINDER, MESH, or SPHERE functions. The objects must be created with default material properties since these properties are controlled with keywords (see below).

Returned Value

result — (n,p,3) byte array containing a 24-bit image of the objects.

Keywords

c — (m,3) array of normalized RGB color components for the objects; by default, c(*,*) = 1.0

k — (m,3,3) array of normalized shade components for the objects: k(i,j,*) contains the ambient, reflective, and transmissive components for c(i,j). The sum of the three components must not exceed one. The default is `k(i,j,*) = [ 0.0, 1.0, 0.0 ]`

v — (3,4) array used to override the view automaticaly generated from !p.t. If defined, *v* works like RENDER's *View* keyword; if undefined. *v* works like RENDER's *Info* keyword.

g — (4,q) array giving position and intensity for q light sources; the sum of the source intensities g(3,*) must equal one. The default is a single light source at the viewer's eye

s — A 2-element vector specifying image size. The default is `[256,256]`

Examples

```
b = LIST( SPHERE(), CYLINDER() )  &  b(1).transform(3,2) = 2
```

```
c = [ [0,0], [1,0], [0,1] ]       &  T3D, /reset, ROTATE=[0,50,0]
TV, RENDER24(b,c=c,s=[500,500]), true=3
```

See Also

POLYSHADE, RENDER

REPLV Function

Standard Library function that replicates a vector into an array.

Usage

result = REPLV(*vector, dim_vector, dim*)

Input Parameters

vector — The vector to be replicated.

dim_vector — A vector specifying the dimensions of the output array.

dim — An integer (≥0) designating the dimension to replicate.

Returned Value

result — An array of dimensions *dim_vector*.

Keywords

None.

Examples

```
PM, REPLV( [0,1], [2,4], 0 )
PM, REPLV( [0,1], [4,2], 1 )
PM, REPLV( [0,1], [4,2], 0 )
```

See Also

EXPAND, REBIN, REPLICATE

RESAMP Function

Standard Library function that resamples an array to new dimensions.

Usage

result = RESAMP(*array*, dim_1, ..., dim_n)

Input Parameters

array — An array of n dimensions.

dim$_i$ — Integers (>0) specifying the new dimensions.

Returned Value

result — The resampled version of *array*.

Keywords

Interp — If set, n-linear interpolation is used instead of the default nearest-neighbor interpolation.

Examples

```
PM, RESAMP( [0,1,2,3], 3 )
PM, RESAMP( [0,1,2,3], 3, /i )
PM, RESAMP( [0,1,2,3], 6, /i )
PM, RESAMP( [[0,1,2,3],[4,5,6,7]], 6, 3, /i )
```

See Also

INTRP, REBIN

SAME Function

Standard Library function that tests if two variables are the same.

Usage

result = SAME(*x*, *y*)

Input Parameters

x — A variable.

y — A variable.

Returned Value

result — 1 if the two variables are the same (within keyword settings), 0 if not.

NOTE For Named Structures, the name is not tested, so {a,x:1} is same as {b,x:1}.

Keywords

NoType — If set, the types of x and y are ignored. (FINDGEN(5) is same as INDGEN(5).)

NoDim — If set, the dimensions of the arrays are ignored. ([1,2,3,4] is same as [[1,2],[3,4]] - same as SAME(a(*),b(*).)

NoVal — If set, the values in the arrays are ignored. ([1,2,3] is same as [3,4,5].)

Examples

To test for exact match:

result = SAME(*a*, *b*)

To test for compatible sizes:

result = SAME(*a*, *b*, */NoType*, */NoVal*)

(Replaces multiple SIZE calls.)

See Also

SIZE

SGN Function

Standard Library function that returns the sign of passed values.

Usage

result = SGN(*x*)

Input Parameters

x — A scalar or array.

Returned Value

result — An integer array of the same size of *x* where each element is:

1 where $x > 0$

-1 where $x < 0$

0 where $x = 0$

Keywords

None.

Examples

```
PM, SGN( [-0.5,0,0.5] )
```

SHIF Function

Standard Library function that shifts an array along one of its dimensions.

Usage

result = SHIF(*array, dimension, shift_amount*)

Input Parameters

array — An array.

dimension — An integer (≥0) designating the shift dimension.

shift_amount — An integer specifying the shift amount.

Returned Value

result — An array of the same dimensions as *array*. *result* is obtained from the input array by a shift of *shift_amount* elements along dimension *dimension*. The shift is not cyclic; elements behind the shift are unaltered.

Keywords

y — If set, the shift is cyclic.

Examples

```
PM, SHIF( [0,1,2,3,4,5], 0, 1 )

PM, SHIF( [0,1,2,3,4,5], 0, -2 )

PM, SHIF( [0,1,2,3,4,5], 0, -2, /y )

PM, SHIF( [[0,1,2],[3,4,5],[6,7,8]], 0, 1 )

PM, SHIF( [[0,1,2],[3,4,5],[6,7,8]], 1, 1 )
```

See Also

SHIFT

SLICE Function

Standard Library function that subsets an array along one of its dimensions.

Usage

result = SLICE(*array, dimension, indices*)

Input Parameters

array — An array.

dimension — An integer (≥0) designating the dimension to subset.

indices — A vector of indices into dimension *dimension*.

Returned Value

result — An array of slices perpendicular to dimension *dimension*.

Keywords

None.

Examples

```
a = INDGEN( 6, 5, 2 )  &  pm, a, ['',''], SLICE( a, 0, [1,3,5] )
```

SMALL_INT Function

Smallest Integer Function. Standard Library function that returns the smallest integer greater than or equal to the passed value. Also known as Ceiling Function.

Usage

result = SMALL_INT(*x*)

Input Parameters

x — An array (scalar).

Returned Value

result — A long array (scalar) of the same dimensions as *x*: *result(i)* is the smallest integer greater than or equal to *x(i)*.

Keywords

None.

Examples

```
PM, SMALL_INT( [-0.5,0,0.5] )


x=[-2.1,-1.9,0,1.9,2.1]
PRINT, SMALL_INT(x)
; -2  -1  0  2  3
```

See Also

FIX, GREAT_INT, NINT

SORTN Function

Standard Library function that sorts an array of n-tuples.

Usage

result = SORTN(*a*)

Input Parameters

a — An (m,n) array of m n-tuples.

Returned Value

result — An array of indices, such that *a(result,*)* is the sorted version of *a*.

Keywords

None.

Examples

```
a = BYTSCL( RANDOMU(seed,9,2), Top=5 )
PM, a
b = SORTN( a )
PM, b
PM, a(b,*)
```

See Also

SORT, UNIQUE, UNIQN

vtkADDATTRIBUTE Procedure

Collects point attributes for VTK datasets.

Usage

vtkADDATTRIBUTE, *attributes*

Input Parameters

attributes — A list variable containing all of the attributes for a dataset. Passing an undefined variable on the first call to creates the initial list. Using the variable on subsequent calls will add elements to the list. You should never have to create or modify the contents of this variable manually.

Keywords

Name — A scalar string specifying a name for this attribute. The default name is the attribute name in lower case.

Lookup_table_name — Only used with scalar attributes. A scalar string specifying the name of the lookup table to be associated with a scalar attribute. The default table is "default."

One and only one of the following keywords can be used to add an attribute of the selected type:

Scalars — A vector of floating point numbers containing scalar values for each entry in points.

Lookup_table — An array of floating point numbers of size (3, *m*) containing normalized RGB values.

Vectors — An array of floating point numbers of size (3, *n*), where *n* equals the number of *Points*, containing the *x*, *y*, and *z* components for each vector.

Normals — An array of floating point numbers of size (3, *n*), where *n* equals the number of points, containing the *x*, *y*, and *z* components for each normal, where the *x*, *y*, and *z* values are normalized to a unit length of 1.

Color_scalars — An array of floating point numbers of size (*m*, *n*), where *n* equals the number of *Points* and *m* is the number of values per color scalar. Values are between 0.0 and 1.0.

Texture_coordinates — An array of floating point numbers of size (*m*, *n*), where *m* is 1, 2, or 3 and *n* equals the number of points.

Tensors — An array of floating point numbers of size (3, 3, *n*), where *n* equals the number of points.

Discussion

This procedure allows a set of attributes to be collected and passed to one of the dataset creation routines: vtkPOLYDATA, vtkSTRUCTUREDPOINTS, vtkSTRUCTUREDGRID, vtkRECTILINIARGRID, or vtkUNSTRUCTUREDGRID. Datasets can have one or more attributes associated with their points, and even more than one attribute of the same type, with the *Name* assigned to the attribute used to distinguish them. For *Scalars*, *Normals*, *Color_scalars*, *Texture_coordinates*, and *Tensors*, the number of supplied attributes must equal the number of points in the dataset to which they will be assigned.

vtkAXES Procedure

Creates a set of axes.

Usage

vtkAXES

Input Parameters

None.

Keywords

Charsize — A floating point scalar or three-element array, the size of the text for tickmark labels and axes labels. (Default: 0.4**Lengths*)

Format — A FORTRAN style format string to use for the tick mark labels. (Default: `'(g10.2)'`)

Name — Specify a name to be used to create this object. If an undefined variable is used or no name specified, then a random name is used. This name can be used in calls to vtkCOMMAND to modify this object.

Position — An array of three floating point numbers in data coordinates describing the origin for the axis. (Default: [0, 0, 0])

Lengths — An array of three floating point numbers describing the length of the *x*, *y*, and *z* axes, respectively, specified in data coordinates. (Default: [1, 1, 1])

Labels — If non-zero, any tick marks drawn are labeled with default values.

LOD — If nonzero, the tickmarks are created as level-of-detail actors to aid in keeping a high frame-rate during frequent render requests due to user mouse interaction. If set to a value greater than 1, the number of points to use in the random cloud.

Sigfig — An integer, the number of significant figures to use for the tick mark labels. (Default: 2).

TextColor — The color to use for text used for [XYZ]Title. See vtkWINDOW (page 85) for possible ways to specify the color. (Default: `'`white`'`)

Tickscale — A float, a scaling value passed to vtkSCATTER for the tick mark glyphs. (Default: 0.33)

Ticksymbol — An integer, passed to vtkSCATTER to set the glyph to use for the tick marks. (Default: 0, a sphere)

Other keywords are listed below. For a description of each keyword, see Chapter 3, Graphics and Plotting Keywords in the *PV*‑*WAVE Reference*.

[XYZ]Tickname	[XYZ]Ticks	[XYZ]Tickv
[XYZ]Title		

Discussion

This procedure creates three axes displayed as lines in the *x*, *y*, and *z* direction, with an optional label at the top of each axis.

Example

```
vtkAxes, lengths = [1,1.5,2]
```

See Also

AXIS

vtkCAMERA Procedure

Changes the camera's parameters.

Usage

vtkCAMERA

Input Parameters

None.

Keywords

Name — Specify a name to be used to create this object. If an undefined variable is used or no name is specified, then a random name is used. This name can be used in calls to vtkCOMMAND to modify this object.

Position — An array of three floating point numbers describing the *x*, *y*, and *z* position for the camera in data coordinates.

FocalPoint — An array of three floating point numbers describing the *x*, *y*, and *z* position for the camera's focal point in data coordinates.

ClippingRange — An array of two floating point numbers describing the distance from the camera to the front and back clipping planes (in data coordinates).

ViewUp — An array of three floating point numbers describing a vector that represents the up direction for the view.

Distance — A floating point number describing the distance from the focal point to the camera (which will modify the *FocalPoint* value) in data coordinates.

ViewAngle — A floating point number that sets the view angle of the camera in degrees.

Azimuth — A scalar value describing the angle in degrees to rotate the camera about the view up vector centered at the focal point. This moves the camera from side to side.

Elevation — A scalar value describing the angle in degrees to rotate the camera about the cross product of the direction of projection and the view up vector centered on the focal point. This moves the camera up and down.

Roll — A scalar value describing the angle in degrees to rotate the camera about the direction of projection. This rolls the camera about the direction of projection.

Discussion

A default camera is created for each vtkWINDOW with these properties: position and focal point such that all objects are visible, with the camera centered on the entire scene; view up along the Y axis; view angle set to 30 degrees; and a clipping range set to `0.1, 1000.0`.

NOTE vtkSURFACE, vtkSCATTER and vtkPOLYSHADE change this default and set the up vector to be along the *z* axis.

Example

```
pyramid_list = [[0,0,0],[0,1,0],[1,0,0],[1,1,0],[.5,.5,1]]
vertex_list=[3,0,2,4,3,2,3,4,3,3,1,4,3,1,0,4,4,0,1,3,2]
   vtkPolyshade, pyramid_list, vertex_list
   vtkCamera, Azimuth=25, Elevation=45, ViewAngle=120
```

vtkCLOSE Procedure

Closes the VTK process.

Usage

vtkCLOSE

Input Parameters

None.

Keywords

None.

Discussion

This procedure closes all VTK windows and shuts down the Tcl/Tk-spawned process. It should be called before exiting PV-WAVE.

Example

A standard close call.

```
vtkCLOSE
```

See Also

vtkINIT, vtkWINDOW, vtkERASE, vtkWDELETE

vtkCOLORBAR Procedure

Adds a color bar legend to a VTK scene using the current PV-WAVE color table.

Usage

vtkCOLORBAR

Keywords

Vertical — If set, the color bar is aligned vertically. Default alignment is horizontal.

***Title* —** The title of the legend. (Default: none)

Position — A three-element array, the position of the lower left corner of the color bar. (Default: [0,0,0])

***NumLabels* —** The number of labels to draw. (Default: 5)

***Width* —** The width of the legend in device coordinates. (Default: 0.8, or 0.15 with /*Vertical*)

***Height* —** The height of the legend in device coordinates. (Default: 0.15, or 0.9 with /*Vertical*)

***CRange* —** A two-element vector, the range of colors (Default: [0,255])

***LRange* —** A two-element vector, the label range (Default: *CRange*)

Sigfig — An integer, the number of significant figures to use for the labels. (Default: 3).

***Name* —** Specifies a name to be used to create this object. If an undefined variable is used or no name is specified, then a random name is used. This name can be used in calls to vtkCOMMAND to modify this object.

NoShadow — If set, labels are drawn without shadows.

vtkCOMMAND Procedure

Sends Tcl and VTK commands to the Tcl process.

Usage

vtkCOMMAND, *command*

Input Parameters

command — A string representing the VTK command to invoke.

Keywords

Result — A string or string array containing any results from the execution of the command in the Tcl shell.

Discussion

The Basic interface to send raw Tcl or VTK commands to the spawned Tcl process.

Example

Use vtkCOMMAND to set the background blue.

```
vtkwindow, 1

vtkcommand, ‘renderer1 SetBackground 0.0 0.0 1.0’

vtkrenderwindow
```

See Also

vtkRENDERWINDOW

vtkERASE Procedure

Erases the contents of the current VTK window.

Usage

vtkERASE [, *background_color*]

Input Parameters

background_color — (optional) The background color to be used for the window, specified as a 24-bit color. See vtkWINDOW (page 85) for possible ways to specify the color.

Keywords

None.

Discussion

This procedure works like ERASE for PV-WAVE windows. It removes all actors, cameras, and lights from the current window.

Example

This example shows vtkERASE removing the axes from the window.

```
vtkwindow, 1

vtkaxes

vtkerase
```

See Also

ERASE, VtkCLOSE, vtkWINDOW

vtkGRID Procedure

Adds 3D grid lines to a VTK scene.

Usage

vtkGRID [, *Number=n*]

Keywords

Number — A scalar or a three element array, the number of segments with grid lines in each (*x*, *y*, *z*) direction. (Default: [1,1,1], a box)

Lengths — A scalar or a three-element array, the extent of the grid. (Default: [1,1,1])

Position — A three-element array, the position of the origin of the grid. (Default: [0,0,0])

Color — The color to use for the polylines (passed to vtkPLOTS). See vtkWINDOW for possible ways to specify the color. (Default: `'`white`'`)

Thick — A float, the thickness of the grid lines (passed to vtkPLOTS). (Default: 1.0)

Name — A string, the name to be used to create this object. If an undefined variable is used or no name is specified, then a random name is used. This name can be used in calls to vtkCOMMAND to modify this object.

UseAxes — If nonzero, the most recently created vtkAXES scale is used to define the Lengths array, which then does not need to be defined explicitly.

LOD — If nonzero, use level-of-detail actors for the grid lines (passed to vtkPLOTS).

Example

```
vtkSURFACE, DIST(20)

vtkGRID, /useAxes, Thick=6

vtkGRID, Number=[2,2,6], Color='red', Thick=4, /useAxes
```

vtkHEDGEHOG Procedure

Creates a HedgeHog (vector) plot.

Usage

vtkHEDGEHOG, *points*, *vectors*, *scalars*

Input Parameters

points — A 3, *n* array of points (location of the lines).

vectors — A 3, *n* array of vectors (orientation and length of the lines).

scalars — (optional) An *n*-element array of scalars (colors of the lines).

Keywords

Scalefactor — A float, a scaling factor to control the size of the oriented lines of the HedgeHog object (Default: 1.0).

SRange — A two-element integer array, the scalar range (Default: [0,255]).

LOD — If nonzero, a level-of-detail actor is created to aid in keeping a high frame-rate during frequent render requests due to user mouse interaction. If set to a value greater than 1, the number of points to use in the random cloud.

***NoRotate* —** Does not perform any camera rotations. Used when a previous call to vtkSURFACE, vtkSCATTER or vtkPOLYSHADE has already set the camera angle.

NoAxes — If set, no axes are created.

NoErase — If nonzero, prevents the window from being erased to the background color before drawing the new scene. If not set, then all lights, cameras, and objects are removed from the scene before objects for the new scene are added.

These keywords are passed to vtkAXES:

Charsize	TextColor	[XYZ]Title	[XYZ]Ticks
[XYZ]Tickv	[XYZ]Tickn	TickScale	TickSymbol
Labels	Sigfig	Format	

See vtkAXES for descriptions.

Example

This example plots a Hanning surface and its normals.

```
LOADCT, 2
n = 50
x = REBIN(INDGEN(n),n,n)
y = TRANSPOSE(x)
z = (n/2)*(HANNING(n,n))
norm = NORMALS(JACOBIAN(LIST(x,y,z)))
;
points = FLTARR(3,n*n,/NoZero)
points(0,*) = x(*)
points(1,*) = y(*)
points(2,*) = z(*)
vectors = FLTARR(3,n*n,/NoZero)
vectors(0,*) = (norm(0))(*)*SQRT(z(*))
vectors(1,*) = (norm(1))(*)*SQRT(z(*))
vectors(2,*) = (norm(2))(*)*SQRT(z(*))
scalars = z(*)
;
vtkHedgeHog, points, vectors, scalars, /NoAxes, scalef=0.75
vtkSurface, REFORM(points(2,*),n,n), $
 Shades=REBIN([0.8,0.8,0.8,.75],4,n,n), $
 /NoErase, /NoRotate, /NoAxes
```

See Also

vtkAXES

vtkINIT Procedure

Initializes the VTK system.

Usage

vtkINIT

Input Parameters

None.

Keywords

File — If set, a temporary file is used to communicate data sets to VTK instead of a socket connection. For very large data sets with many floating-point values, this method is considerably faster; however, read/write permissions are required. If set to one (`/File`), any data set greater than 1024 bytes is written to file. If set to a value (`File=`*fbytes*), data sets larger than *fbytes* are written to file; smaller data sets are sent via sockets. Setting /*File* is equivalent to setting File=1024. This keyword affects only data: commands are always sent by the socket connection.

***Noshell* —** If set, the keyword is passed along to the SPAWN procedure that initiates the VTK Tcl process. This keyword is required when calling VTK routines from a JWAVE wrapper and should not be used otherwise.

Path — Used in conjunction with the *File* keyword, a string indicating the file path to the directory where the temporary file(s) are to be created.

Print — If present and nonzero, causes the output from the spawned Tcl/Tk shell to be sent back to PV-WAVE and displayed in the console. This keyword is useful for debugging low-level VTK calls.

Timeout — A floating point scalar specifying a time interval in seconds which vtkINIT will wait before giving up on establishing a socket connection to the spawned Tcl shell. (Default: 20)

Discussion

This procedure must be performed before any other VTK commands. It causes a Tcl/Tk shell to be spawned and sets up communication with it. It also initialized various internal VTK parameters. The following routines will automatically call vtkINIT if it has not already been called: vtkWINDOW, vtkPOLYSHADE, vtk-SURFACE, and vtkSCATTER.

Example

This example uses vtkINIT to initialize VTK with a timeout of 10 seconds.

```
vtkINIT, timeout=10
```

See Also

VtkCLOSE

vtkLIGHT Procedure

Adds a light to a VTK window.

Usage

vtkLIGHT

Input Parameters

None.

Keywords

Name — Specifies a name to be used to create this object. If an undefined variable is used or no name specified, then a random name is used. This name can be used in calls to vtkCOMMAND to modify this object.

Color — The color for the light. See vtkWINDOW (page 85) for possible ways to specify the color. (Default: `'`white`'`)

Position — An array of three floating point numbers describing the *x*, *y*, and *z* position for the light. The default behavior is to have the light follow the camera position.

FocalPoint — An array of three floating point numbers describing the *x*, *y*, and *z* position for the light's focal point.

Intensity — A float value between 0.0 and 1.0 specifying the intensity of the light.

DirectionAngle — An array of two floating point numbers that set the position and focal point of a light based on elevation and azimuth. The light is moved to shine from the given angle. Angles are given in degrees.

Discussion

A white light, which follows the camera position, is created by default for a VTK Window.

NOTE Other light parameters not supported in this wrapper can be set using the assigned *Name* and vtkCOMMAND.

Example

```
pyramid_list = [[0,0,0],[0,1,0],[1,0,0],[1,1,0],[.5,.5,1]]
vertex_list=[3,0,2,4,3,2,3,4,3,3,1,4,3,1,0,4,4,0,1,3,2]
vtkPolyshade, pyramid_list, vertex_list
     vtkLight, Color='0000FF'XL, Position=[10,10,10]
     vtkLight, Color='00FF00'XL, Position=[10,10,-10]
     vtkLight, Color='FF0000'XL, Position=[-10,-10,-10]
```

vtkPLOTS Procedure

Adds a polyline.

Usage

vtkPLOTS, *points*

Input Parameters

points — An array of floating point numbers of size (3, *n*) where *n* is the number of points. `points(0,*)` is taken as an *x* value, `points(1,*)` is taken as a *y* value, and `points(2,*)` is taken as a *z* value.

Keywords

Name — Specifies a name to be used to create this object. If an undefined variable is used or no name specified, then a random name is used. This name can be used in calls to vtkCOMMAND to modify this object.

Color — The color to use for the polyline. See vtkWINDOW (page 85) for possible ways to specify the color. (Default: `'white'`)

Thick — A float describing the thickness of the polyline. The default is 1.0, which is scaled to correspond to a radius of 0.001 in data coordinates.

LOD — If nonzero, a level-of-detail actor is created to aid in keeping a high frame-rate during frequent render requests due to user mouse interaction. If set to a value greater than 1, the number of points to use in the random cloud.

Nolines — If nonzero, causes a cloud of points to be displayed rather than a polyline.

Discussion

This procedure is similar to the PLOTS procedure for PV-WAVE windows. Polylines are drawn as connected cylinders.

Example

```
z = findgen(1000)/999

x = z*sin(50*z)

y = z*cos(50*z)

vtkplots, transpose([[x],[y],[z]]), thick=5, color='blue'

vtkwindow,2

vtkplots, transpose([[x],[y],[z]]), color='red',/nolines
```

See Also

PLOTS, vtkSCATTER

vtkPOLYDATA Procedure

Passes vertex/polygon lists, lines, points, and triangles to VTK.

Usage

vtkPOLYDATA, *points*

Input Parameters

points — A two-dimensional array of floating point numbers of size (3, *n*) describing *x*, *y*, and *z* points.

Keywords

Restore — An associative array containing all of the data and attributes for a polydata dataset, usually created using the *Save* keyword. If this parameter is passed, then only the keywords *Name* and *Filename* can be used.

Name — Specifies a name to be used to create this data source. This name can be used in calls to vtkCOMMAND.

Filename — File to store data using standard VTK ASCII format.

Save — Returns the data in the specified variable stored in an associative array. Data is *not* sent to VTK if this parameter is specified.

Polygons — A vector of integers describing polygons, organized as vertex count followed by indices into *points*, repeated for all polygons. This is the same format as a polygon list used in POLYSHADE.

Vertices — A vector of integers describing vertices, organized as vertex count followed by indices into *points*, repeated for all vertices.

Lines — A vector of integers describing polylines, organized as vertex count followed by indices into *points*, repeated for all polylines.

Triangle_Strips — A vector of integers describing triangle strips, organized as vertex count followed by indices into *points*, repeated for all triangle strips.

Attributes — A list created using vtkADDATTRIBUTE containing one or more attributes associated with the points in the dataset.

Discussion

Contains points and polygons (like the polygon vertex list in PV‑WAVE) as well as vertices, lines, and triangle strips. See the VTK documentation, which can be downloaded from *http://public.kitware.com*, for more details on the data and attributes for the PolyData dataset format.

vtkPOLYSHADE Procedure

Renders a polygon object.

Usage

vtkPOLYSHADE, *vertices*, *polygons*

Input Parameters

Vertices — A (3, *n*) array containing the *x*-, *y*-, and *z*-coordinates of each vertex in data coordinates.

Polygons — An integer or longword array containing the indices of the vertices of each polygon. The vertices of each polygon should be listed either clockwise or counterclockwise order when observed from outside the surface. The vertex description of each polygon is a vector of the form [*n*, *i*0, *i*1, ... , *in* - 1], and the array polygons is the concatenation of the lists of each polygon.

Keywords

Name — Specify a name to be used to create this object. If an undefined variable is used or no name specified then a random name is used. This name can be used in calls to vtkCOMMAND to modify this object.

Color — An array expression, of the same dimensions as the number of vertices passes (the value "*n*" above), containing the color index at each vertex. Alternately an expression describing one color to be used for the entire polygonal surface or wireframe. If this keyword is omitted, a white surface is displayed.

To specify a single color, see vtkWINDOW (page 85) for possible ways to specify the color. If a two-dimensional array of colors is specified (for an image overlay) the shades variable can be in any of these formats:

FIX(*n*)	A one-dimensional array of short integers or bytes specifying an index into the current PV‑WAVE color table for each point. The RGB color for each point is obtained from the corresponding entry in the current PV‑WAVE color table.
LONG(*n*)	A one-dimensional array of long integers specifying the 24-bit color at each point.
FLOAT(3, *n*)	A floating point array of size (3, *n*) containing the normalized values specifying the red, green, and blue components of the color at each point.
FLOAT(4, *n*)	A floating point array of size (4, *n*) containing the normalized values specifying the red, green, blue, and alpha components of the color at each vertex. The alpha component is the transparency where 0.0 is completely transparent and 1.0 is opaque.

Wireframe — When present and nonzero, a wire-frame mesh is drawn rather than a shaded surface.

LOD — If nonzero, a level-of-detail actor is created to aid in keeping a high framerate during frequent render requests due to user mouse interaction. If set to a value greater than 1, the number of points to use in the random cloud.

NoAxes — If set, no axes are created.

NoRotate — Does not perform any camera rotations. Used when a previous call to vtkSURFACE, vtkSCATTER or vtkPOLYSHADE has already set the camera angle.

NoErase — If nonzero prevents the window from being erased to the background color before drawing the new scene. If not set then all lights, cameras, and objects are removed from the scene before objects for the new scene are added.

These keywords are passed to vtkAXES:

[XYZ]Ticks	[XYZ]Tickv	[XYZ]Tickn	TickScale
TickSymbol	Labels	Sigfig	Format

Other keywords are listed below. For a description of each keyword, see Chapter 3, *Graphics and Plotting Keywords* in the *PV‑WAVE Reference*.

Ax	Az

Discussion

This procedure is similar to the POLYSHADE procedure for PV-WAVE windows. Wireframes can be produced as well as surfaces shaded in one color or overlaid with an image. Transparency is also supported.

Example

```
pyramid_list = [[0,0,0],[0,1,0],[1,0,0],[1,1,0],[.5,.5,1]]
vertex_list=[3,0,2,4,3,2,3,4,3,3,1,4,3,1,0,4,4,0,1,3,2]
vtkPolyshade, pyramid_list, vertex_list, color='blue'
```

See Also

AXIS, POLYSHADE

vtkPPMREAD Function

Reads a PPM file.

Usage

image = vtkPPMREAD (*filename*)

Input Parameters

filename — File path of PPM file.

Returned Value

image — An array (3, *width*, *height*) of bytes containing the 24-bit image.

Keywords

None.

Discussion

This function is used to read the rudimentary PPM binary files created by VTK and containing images stored as RGB values. The images can be displayed with `TV, image, True=1` or converted to an 8-bit image using `ipcolor_24_8`.

Example

Example of reading the file `wave.ppm` and storing an image.

```
Image=vtkppmread('wave.ppm')
```

See Also

vtkPPMWRITE, vtkTVRD

vtkPPMWRITE Procedure

Writes the contents of a VTK window to a PPM file.

Usage

vtkPPMWRITE [, *window_index*]

Input Parameters

window_index — (optional) An integer specifying the index of an existing VTK window. If omitted, the current window is used.

Keywords

Filename — File path to store PPM file. (Default: `'wave.ppm'`)

Discussion

This procedure saves a snapshot of the selected window as a PPM file. It is important to make sure the VTK window fully visible when this routine is called because an obscuring portion of another window will be captured as part of the image. This is a limitation of VTK.

Example 1

Writing a PPM file.

```
vtkwindow, 1
vtkaxes
vtkPPMWRITE, 1
```

Example 2

Writing the PPM file to the file name of `PV.ppm`.

```
vtkwindow, 2, background='blue'
vtkPPMWRITE, 2, filename='PV.ppm'
```

See Also

vtkPPMREAD, vtkTVRD

vtkRECTILINEARGRID Procedure

Passes data describing a rectilinear grid to VTK.

Usage

vtkRECTILINEARGRID, *Dimensions*

Input Parameters

Dimensions — A 3-element vector of integers describing dimensions in *x*, *y*, and *z*. Use `1` for the third dimension if only a two-dimensional array is described.

Keywords

Restore — An associative array containing all of the data and attributes for a polydata dataset, usually created using the *Save* keyword. If this parameter is passed, then only the keywords *Name* and *Filename* can be used.

Name — Specifies a name to be used to create this data source. This name can be used in calls to vtkCOMMAND.

Filename — File to store data using standard VTK ASCII format.

Save — Returns the data in the specified variable stored in an associative array. Data is *not* sent to VTK if this parameter is specified.

X_coordinates — A vector of floating point numbers of the same length as the first dimension in *Dimensions* describing monotonically increasing coordinate values. Increasing integers starting with 0 are used as a default.

Y_coordinates — A vector of floating point numbers of the same length as the second dimension in *Dimensions* describing monotonically increasing coordinate values. Increasing integers starting with 0 are used as a default.

Z_coordinates — A vector of floating point numbers of the same length as the third dimension in *Dimensions* describing monotonically increasing coordinate values. Increasing integers starting with 0 are used as a default.

Attributes — A list created using vtkADDATTRIBUTE containing one or more attributes associated with the points in the dataset.

Discussion

This procedure creates a dataset with a regular topology and semiregular geometry aligned along the *x*, *y*, and *z* axes. See the VTK documentation, which can be downloaded from *http://public.kitware.com*, for more details on the data and attributes for the RectiliniarGrid dataset format.

vtkRENDERWINDOW Procedure

Renders a VTK window.

Usage

vtkRENDERWINDOW [, *window_index*]

Input Parameters

window_index — (optional) An integer specifying the index of an existing VTK window. If omitted, the current window is used.

Keywords

None.

Discussion

Call this procedure after all objects (lights, cameras, surfaces, polygon meshes, etc.) have been added to the window. This routine starts the rendering process and creates the initial rendered scene. You need to call this procedure only if you used the *Norender* keyword with vtkWINDOW, or if you are making low-level calls to VTK using vtkCOMMAND.

Example 1

This example shows the essense of vtkRENDERWINDOW.

```
vtkwindow, 1, /norender

vtkaxes

vtkrenderwindow
```

Example 2

A more complicated example:

```
vtkwindow, 2, /norender

V=[[0,0,0],[1,0,0],[1,1,0],[0,1,0]]

p=[4,0,1,2,3]

vtkpolyshade, v, p

vtkaxes

vtktext, ‘This is VTK’, charsize=[1.0,1.0,1.0], color=’blue’

vtkrenderwindow, 2
```

See Also

vtkWINDOW, vtkCOMMAND

vtkSCATTER Procedure

Renders 3D points.

Usage

vtkSCATTER, *points*

Input Parameters

points — A float array of size (3, *n*) describing *x*, *y*, and *z* points in data coordinates.

Keywords

Name — Specifies a name to be used to create this object. If an undefined variable is used or no name specified, then a random name is used. This name can be used in calls to vtkCOMMAND to modify this object.

Symbol — A scalar integer describing the type of marker or glyph to be displayed for each point. Supported values are 0-4 where:

0 = Sphere

1 = Cube

2 = Cone

3 = Cylinder

4 = Earth

Color — An array expression of the same dimensions as the number of points (the value *n* above), containing the color index at each point. Alternately an expression describing one color to be used for all points. If this keyword is omitted, white points are displayed.

To specify a single color, see vtkWINDOW (page 85). If a vector of colors is specified, the color variable can be in any of these formats:

FIX(*n*)	A one-dimensional array of short integers or bytes specifying an index into the current PV‑WAVE color table for each point. The RGB color for each point is obtained from the corresponding entry in the current PV‑WAVE color table.

LONG(*n*)	A one-dimensional array of long integers specifying the 24-bit color at each point.
FLOAT(3, *n*)	A floating point array of size (3, *n*) containing the normalized values specifying the red, green, and blue components of the color at each point.

Scale — A float value specifying the scaling factor for the size of each glyph. (Default: 1.0)

LOD — If nonzero, the glyphs are created as level-of-detail actors to aid in keeping a high frame-rate during frequent render requests due to user mouse interaction. If set to a value greater than 1, the number of points to use in the random cloud.

NoRotate — Does not perform any camera rotations. Used when a previous call to vtkSURFACE, vtkSCATTER or vtkPOLYSHADE has already set the camera angle.

NoErase — If nonzero, prevents the window from being erased to the background color before drawing the new scene. If not set, then all lights, cameras, and objects are removed from the scene before objects for the new scene are added.

NoAxes — If present and non zero, then no *x*, *y*, and *z* axes will be drawn.

TextColor — The color for text used for the axes titles. See vtkWINDOW (page 85) for possible ways to specify the color. (Default: `'white'`)

These keywords are passed to vtkAXES:

[XYZ]Ticks	[XYZ]Tickv	[XYZ]Tickn	TickScale
TickSymbol	Labels	Sigfig	Format

Other keywords are listed below. For a description of each keyword, see Chapter 3, *Graphics and Plotting Keywords* in the *PV-WAVE Reference*.

Ax	Az	Charsize
[XYZ]Title[	[XYZ]Range	

Discussion

This procedure plots points in space using three-dimensional markers (glyphs) with optional axes.

Example

```
data=FLTARR(3,200)
        ; Create the data array.
s=DC_READ_FREE(!Data_Dir+'scattered.dat',data,/Column,/Resize)
        ; Read in the data
vtkSCATTER,data
        ; A quick look at the data.
TEK_COLOR
        ; Set up a color table.
s=SIZE(data) & c=INDGEN(s(2))
        ; Create an array of color values for the points.
     vtkSCATTER,data,Color=c,Xtitle='X Values',Ytitle='Y $
    Values', $
    Ztitle='Z Values',TextColor=16,ax=10,az=100,Scale=.6
        ; A better look at the data.
```

See Also

AXIS   vtkPLOTS

vtkSLICEVOL Procedure

Creates a sliced 3D volume at specific *x*, *y*, *z* locations.

Usage

vtkSLICEVOL, *v*, [sx=*sx*, sy=*sy*, sz=*sz*, xc=*xc*, yc=yc, zc=*zc*]

Input Parameters

v — A 3D array, the volume to slice.

Keywords

sx — A 1D array, the *x* coordinate(s) at which to slice the volume.

sy — A 1D array, the *y* coordinate(s) at which to slice the volume.

sz — A 1D array, the *z* coordinate(s) at which to slice the volume.

xc — A 1D array with the same number of elements as the first dimension of *v*, the *x* coordinates of the volume.

yc — A 1D array with the same number of elements as the second dimension of *v*, the *y* coordinates of the volume.

zc — A 1D array with the same number of elements as the third dimension of *v*, the *z* coordinates of the volume.

Interp — If set, the shading is interpolated (passed to RESAMP).

Dim — An integer, the number of vertices on each side of each plane. (Default: 25)

Name — Specifies a name to be used to create this object. If an undefined variable is used or no name specified, then a random name is used. This name can be used in calls to vtkCOMMAND to modify this object.

Wireframe — When present and nonzero, a wire-frame mesh is drawn rather than a shaded surface.

LOD — If nonzero, a level-of-detail actor is created to aid in keeping a high frame-rate during frequent render requests due to user mouse interaction. If set to a value greater than 1, the number of points to use in the random cloud.

NoRotate — Does not perform any camera rotations. Used when a previous call to vtkSURFACE, vtkSCATTER or vtkPOLYSHADE has already set the camera angle.

NoErase — If nonzero, prevents the window from being erased to the background color before drawing the new scene. If not set, then all lights, cameras, and objects are removed from the scene before objects for the new scene are added.

Other keywords are listed below. For a description of each keyword, see Chapter 3, *Graphics and Plotting Keywords* in the *PV-WAVE Reference*.

Ax	Az

Discussion

If no slices are requested through the *sx*, *sy*, and *sz* keywords, the volume is sliced at the midpoints of each index.If *xc*, *yc*, or *zc* are not provided, indices into *v* are used.

Example

```
x = genvect(-5,5,.25)

y = genvect(-4,4,.2)
```

```
z = genvect(-3,3,.15)
v = sqrt(TENSOR_ADD(TENSOR_ADD(x^2,0.3*y^2),1.5*z^2))
vtkSliceVol, v, sx=[-4,.4,2.6], sy=-.15, sz=[-3,1], $
 xc=x, yc=y, zc=z, dim=15
```

See Also

SLICE

vtkSTRUCTUREDGRID Procedure

Passes data describing a structured grid to VTK.

Usage

vtkSTRUCTUREDGRID, *dimensions*, *points*

Input Parameters

dimensions — A 3-element vector of integers describing dimensions in *x*, *y*, and *z*. Use “1” for the third dimension if only a two-dimensional array is described.

points — A two-dimensional array of floating point numbers of size (3, *n*) describing *x*, *y*, and *z* points.

Keywords

Restore — An associative array containing all of the data and attributes for a polydata dataset, usually created using the *Save* keyword. If this parameter is passed, then only the keywords *Name* and *Filename* can be used.

Name — Specifies a name to be used to create this data source. This name can be used in calls to vtkCOMMAND.

Filename — File to store data using standard VTK ASCII format.

Save — Returns the data in the specified variable stored in an associative array. Data is *not* sent to VTK if this parameter is specified.

Attributes — A list created using vtkADDATTRIBUTE containing one or more attributes associated with the points in the dataset.

Discussion

1, 2, or 3D point data on a topological grid, where the actual points are specified as *x*, *y*, and *z* values in Cartesian coordinates. See the VTK documentation, which can be downloaded from *http://public.kitware.com*, for more details on the data and attributes for the StructuredGrid dataset format.

vtkSTRUCTUREDPOINTS Procedure

Passes data describing structured points to VTK.

Usage

vtkSTRUCTUREDPOINTS, *dimensions*

Input Parameters

dimensions — A 3-element vector of integers describing dimensions in *x*, *y*, and *z*. Use `1` for the third dimension if only a two-dimensional array is described.

Keywords

Restore — An associative array containing all of the data and attributes for a polydata dataset, usually created using the *Save* keyword. If this parameter is passed then only the keywords *Name* and *Filename* can be used.

Name — Specifies a name to be used to create this data source. This name can be used in calls to vtkCOMMAND.

Filename — File to store data using standard VTK ASCII format.

Save — Returns the data in the specified variable stored in an associative array. Data is NOT sent to VTK if this parameter is specified.

Origin — A 3-element vector of floating point numbers containing the *x*, *y*, and *z* origin point for the data.

Spacing — A 3-element vector of floating point numbers containing the spacing (width, height, length) of the cubical cells that compose the data set.

Attributes — A list created using vtkADDATTRIBUTE containing one or more attributes associated with the points in the dataset.

Discussion

Definition of a 1, 2, or 3D arrays (describing lines, grids and voxels), their origin, and spacing. See the VTK documentation, which can be downloaded from *http://public.kitware.com*, for more details on the data and attributes for the Structured-Points dataset format.

vtkSURFACE Procedure

Renders a surface.

Usage

vtkSURFACE, *z* [,*x*] [,*y*]

Input Parameters

z — A two-dimensional array containing the values that describe the surface. If *x* and y are supplied, the surface is plotted as a function of the *x* and *y* locations specified by their contents. Otherwise, the surface is generated as a function of the array index of each element of *z*.

x — (optional) A vector or two-dimensional array specifying the *x*-coordinates for the surface.

If *x* is a vector, each element of *x* specifies the *x*-coordinate for a column of *z*. For example, *x*(0) specifies the *x*-coordinate for *z*(0, *).

If x is a two-dimensional array, each element of *x* specifies the *x*-coordinate of the corresponding point in *z* (*xij* specifies the *x*-coordinate for *zij*).

y — (optional) A vector or two-dimensional array specifying the *y*-coordinates for the surface.

If *y* is a vector, each element of *y* specifies the y coordinate for a row of *z*. For example, *y*(0) specifies the y-coordinate for *z* (*, 0).

If *y* is a two-dimensional array, each element of y specifies the *y*-coordinate of the corresponding point in *z* (*yij* specifies the *y*-coordinate for *zij*).

Keywords

Name — Specifies a name to be used to create this object. If an undefined variable is used or no name specified, then a random name is used. This name can be used in calls to vtkCOMMAND to modify this object.

Shades — An array expression of the same dimensions as *z*, containing the color index at each point. Alternately, an expression describing one color to be used for the entire surface or wireframe. If this keyword is omitted, a white surface is displayed.

To specify a single color, see vtkWINDOW (page 85). If a two-dimensional array of colors is specified (for an image overlay), the shades variable can be in any of these formats:

***Fix*(*x, y*)** — A two-dimensional array of short integers or bytes specifying an index into the current PV-WAVE color table for each point. The RGB color for each point is obtained from the corresponding entry in the current PV-WAVE color table.

***Long*(*x, y*)** — A two-dimensional array of long integers specifying the 24-bit color.

***Float*(*3, x, y*)** — A floating point array of size (3, *x*, *y*) containing the normalized values specifying the red, green, and blue components of the color at each point.

***Float*(*4,x*)** — A floating point array of size (4, *x*, *y*) containing the normalized values specifying the red, green, blue, and alpha components of the color at each point. The alpha component is the transparency where 0.0 is completely transparent and 1.0 is opaque.

Wireframe — When present and nonzero, a wire-frame mesh is drawn rather than a shaded surface.

LOD — If nonzero, a level-of-detail actor is created to aid in keeping a high framerate during frequent render requests due to user mouse interaction. If set to a value greater than 1, the number of points to use in the random cloud.

NoRotate — Does not perform any camera rotations. Used when a previous call to vtkSURFACE, vtkSCATTER or vtkPOLYSHADE has already set the camera angle.

NoAxes — If present and non zero, then no *x*, *y*, or *z* axes will be drawn.

NoErase — If nonzero, prevents the window from being erased to the background color before drawing the new scene. If not set, then all lights, cameras, and objects are removed from the scene before objects for the new scene are added.

TextColor — The color to use for text used for the axes titles. See vtkWINDOW for possible ways to specify the color. (Default: `'white'`)

These keywords are passed to vtkAXES:

[XYZ]Ticks	[XYZ]Tickv	[XYZ]Tickn	TickScale
TickSymbol	Labels	Sigfig	Format

Other keywords are listed below. For a description of each keyword, see Chapter 3, *Graphics and Plotting Keywords* in the *PV‑WAVE Reference*.

Ax	Az	Charsize
[XYZ]Title[	[XYZ]Range	

Discussion

This procedure is similar to the SURFACE and SHADE_SURF procedures for PV‑WAVE windows. Wireframes can be produced as well as surfaces shaded in one color or overlaid with an image. Transparency is also supported.

Example

This example demonstrates a surface created in a regular PV‑WAVE window compared to one using vtkSURFACE.

```
pikes=FLTARR(60,40)
s=DC_READ_FREE(!Data_Dir+'pikeselev.dat',pikes)
    ; Read in the data values for elevation.
snow=FLTARR(60,40)
s=DC_READ_FREE(!Data_Dir+'snowpack.dat',snow)
    ; Read in the data for snowpack
loadct,5
    ; Load a color table.
surface,pikes
    ; Create a wiremesh surface using a regular PV-WAVE window.
vtksurface,pikes/250,/wireframe
    ; Create a wiremesh surface using the VTK toolkit.  Notice that the toolkit doesn't scale
    ; the data for you so in order to make sense out of the resulting graphic you need to
    ; scale the data yourself, in this example it was done by dividing by 250.
shade_surf,pikes,shades=bytscl(snow)
    ; Create a shaded surface using a regular PV-WAVE window.
vtksurface,pikes/250,shades=bytscl(snow)
```

```
; Create a shaded surface using the VTK toolkit.  As above, the user does the scaling
; of the data.
```

See Also

AXIS,   vtkAXES

vtkSURFGEN Procedure

Generates a 3D surface from sampled points assumed to lie on a surface.

Usage

vtkSURFGEN, *points*

Input Parameters

points — A 3, *n* array of points that lie on a surface.

Keywords

Reverse — By default, the normals of the computed surface are inward facing. If outward normals are required, set this keyword.

Neighbors — An integer, the number of neighbors each points has. Use a larger value if the spread of points is not even. (Default: 20)

Spacing — A float, the spacing of the 3D sampling grid. If not set, the VTK class makes a reasonable guess.

Filename — An ASCII VTK file to create containing the dataset generated by the VTK filter.

Data — (Output). A returned associative array containing two keys: data(“POINTS”) is a 3, *n* float array containing the points generated by the VTK filter and data(“VERTICES”) is an *n*+1 element long array containing topology information for the generated dataset. A filename must be defined to have data returned by this keyword.

Name — A string, the name to be used to create this object. If an undefined variable is used or no name is specified, then a random name is used. This name can be used in calls to vtkCOMMAND to modify this object.

Color — The color to use for the generated surface. See vtkWINDOW for possible ways to specify the color. (Default: `'white'`)

Wireframe — When present and nonzero, a wire-frame mesh is drawn rather than a shaded surface.

LOD — If nonzero, a level-of-detail actor is created to aid in keeping a high frame-rate during frequent render requests due to user mouse interaction. If set to a value greater than 1, the number of points to use in the random cloud.

This routine also accepts these keywords to control the initial camera and the axes (see vtkSURFACE and vtkAXES):

[XYZ]Range	Ax	Az	NoRotate
NoAxes	NoErase	Charsize	TextColor
[XYZ]Title	[XYZ]Ticks	[XYZ]Tickv	[XYZ]Tickn
Tickscale	Ticksymbol	Labels	Format
Sigfig			

Example

```
@math_startup
s = TRANSPOSE(random(100, /Sphere, Parameter=3))
vtkSURFGEN, s, /NoAxes
vtkSCATTER, s, /NoAxes, Symb=1, Color='red', Scale=0.5, $
 /NoRotate, /NoErase
```

See Also

vtkAXES, vtkSURFACE

vtkTEXT Procedure

Adds a text string.

Usage

vtkTEXT, *string*

Input Parameters

string — The scalar string containing the text that is to be output to the display surface. If not of string type, it is converted prior to use.

Keywords

Name — Specifies a name to be used to create this object. If an undefined variable is used or no name specified, then a random name is used. This name can be used in calls to vtkCOMMAND to modify this object.

Position — An array of three floating point numbers specifying the *x*, *y*, and *z* position for the beginning of text. (Default: [0,0,0])

Color — The color to use for text. See vtkWINDOW (page 85) for possible ways to specify the color. (Default: `'`white`'`)

Follow — If nonzero, forces the text to always be facing the camera.

Orientation — An array of three floating point numbers specifying the *x*, *y*, and *z* rotations of text in degrees. The actual rotations are performed in this order: *z* then *x* and finally *y*. This keyword has no effect if *Follow* is specified.

Keyword Charsize is also supported. For a description, see Chapter 3, *Graphics and Plotting Keywords* in the *PV-WAVE Reference*.

Discussion

This procedure is similar to the XYOUTS procedure for PV-WAVE windows.

Example

```
vtkText, "This is vtkText", color="red", charsize=10
```

vtkTVRD Function

Returns the contents of a VTK window as a bitmapped image.

Usage

image = vtkTVRD([*window_index*])

Input Parameters

window_index — (optional) An integer specifying the index of an existing VTK window. If omitted, the current window is used.

Keywords

Filename — File path to store a temporary PPM file. Default is "`wave.ppm`."

Returned Value

image — An array (3, *width*, *height*) of bytes containing the 24-bit image.

Discussion

This function works like TVRD for PV-WAVE windows. It uses vtkPPMWRITE and vtkPPMREAD to save and then read the contents of a window. The temporary file created is deleted when done.

Example

```
vtkWINDOW, 7

vtkAXES

tv, vtkTVRD(7), /TRUE
```

See Also

TVRD, vtkPPMWRITE, vtkPPMREAD

vtkUNSTRUCTUREDGRID Procedure

Passes data describing an unstructured grid to VTK.

Usage

vtkUNSTRUCTUREDGRID, *points*, *cells*, *cell_types*

Input Parameters

points — A two-dimensional array of floating point numbers of size (3, *n*) describing *x*, *y*, and *z* points.

cells — A Vector of integers describing cells, organized as vertex count followed by indices into Points, repeated for all cells.

Cell_types — A Vector of integers describing the cell type for each cell. Valid types are values between 1-12.

Keywords

Restore — An associative array containing all of the data and attributes for a polydata dataset, usually created using the *Save* keyword. If this parameter is passed, then only the keywords *Name* and *Filename* can be used.

Name — Specifies a name to be used to create this data source. This name can be used in calls to vtkCOMMAND.

Filename — File to store data using standard VTK ASCII format.

Save — Returns the data in the specified variable stored in an associative array. Data is NOT sent to VTK if this parameter is specified.

Attributes — A list created using vtkADDATTRIBUTE containing one or more attributes associated with the points in the dataset.

Discussion

Arbitrary combinations of twelve (12) cell types, ranging from points, lines, polygons to voxels. See the VTK documentation, which can be downloaded from *http://public.kitware.com*, for more details on the data and attributes for the UnStructuredGrid dataset format.

vtkWDELETE Procedure

Closes a VTK window without shutting down the Tcl process.

Usage

vtkWDELETE [, *window_index*]

Input Parameters

window_index — (optional) An integer specifying the index of an existing VTK window. If omitted, the current window is used.

Keywords

All — If nonzero, causes all VTK windows to be closed.

Discussion

This procedure works like WDELETE for PV-WAVE windows. It closes an individual VTK window but does not shut down the Tcl process. Use vtkCLOSE to close all windows and shut down the spawned Tcl process.

Example 1

Deleting a window.

```
vtkwindow, 1
vtkwdelete
```

Example 2

```
vtkwindow, 1
vtkwindow,2
vtkwindow,3
vtkwdelete, /all
```

See Also

vtkCLOSE, vtkWINDOW

vtkWINDOW Procedure

Creates a VTK window.

Usage

vtkWINDOW [,*window_index*]

Input Parameters

window_index — (optional) An integer specifying the index of the newly created window.

If *window_index* is omitted, 0 is used as the index of the new window.

If the value of *window_index* specifies an existing window, the existing window is deleted and a new window is created.

Keywords

Free — If nonzero, creates a window using an unused window index. This keyword can be used instead of specifying the *window_index* parameter.

NoRender — If nonzero, prevents individual objects (vtkLIGHT, vtkAXES, vtkPOLYSHADE, etc.) from being rendered as they are added to the window. Specifying *NoRender* can speed up the initial display of a scene if you have multiple objects in it. If specified, you must manually call vtkRENDERWINDOW after you have added all of your objects to the window.

Background — Background color for the window. The color can be specified in any of the following ways (the color ‘red’ is used here as an example):

‘red’	See the file `<`*vni*`>/vtk-3_2/lib/vtkcolornames.pro` for a complete list of supported color names, where <*vni*> is the path to the PV-WAVE installation.
‘FF0000’XL	A long integer hexadecimal value specifying the 24-bit color.
[1.0, 0.0, 0.0]	A three-element vector of normalized floating point values specifying the red, green, and blue components of the color.

2	If a short byte or short integer value is passed, the RGB color is obtained from the corresponding entry in the current PV‑WAVE color table. In this case, when TEK_COLOR has been called, color index 2 is red.

/NoInteract — If nonzero, indicates that you do not wish to provide the standard set of mouse controls for viewing the 3D scene. The resulting scene can be manipulated only by programmatically setting the positional parameters for objects or cameras.

XPos, ***YPos*** — The *x* and *y* positions of the lower-left corner of the new window, specified in device coordinates.

If no position is specified, a position of (0,0) is used.

XSize — The width of the window, in pixels. (Default: 400)

YSize — The height of the window, in pixels. (Default: 400)

Discussion

This procedure is similar to the PV‑WAVE WINDOW command. It allows the created window to have built-in interaction associated with it.

Example 1

This example shows how to bring up a VTK window.

```
vtkWINDOW, 1
```

Example 2

This example shows how to bring up a VTK window with a blue background and with the mouse controls disabled. *windownum* is the number of the free window.

```
vtkWINDOW, windownum, /Free, background='blue', /nointeract
```

See Also

vtkRENDERWINDOW, vtkCLOSE, vtkWDELETE, vtkERASE, vtkWSET

vtkWRITEVRML Procedure

Creates a Virtual Reality Modeling Language file (VRML .wrl file) from a scene in a VTK window.

Usage

vtkWRITEVRML*, filename* [, *WindowID=id*, *Speed=s*]

Input Parameters

Filename — A string, the file to write (should end in ".wrl").

Keywords

Windowid — An integer, the VTK window to use as the source. (Default: the currently active VTK window)

Speed — A float, the navigation speed. (Default: 4.0)

Discussion

To view a VRML .wrl file in a browser, you need a plug-in. Consult the FAQ at http://www.vrml.org for current information and to obtain a plug-in.

See Also

VRML Routines

vtkWSET Procedure

Sets the active VTK window.

Usage

vtkWSET [, *window_index*]

Input Parameters

window_index — (optional) An integer specifying the index of an existing VTK window. If omitted, the current window is used.

Keywords

None.

Discussion

This procedure works like WSET for PV-WAVE windows, which is used to select the current, or "active" window to be used by the VTK routines.

Example

Setting the window to the first window opened.

```
vtkwindow, 1
vtkwindow, 2
vtkwindow, 3
vtkaxes
vtkwset, 1
vtkaxes
```

See Also

vtkWINDOW, WSET

WgOrbit Procedure

Creates an interactive window for viewing objects.

Usage

WgOrbit, *vertices*, *polygons*, *parent*, *shell*

Input Parameters

vertices — A (3,n) array of points on the surfaces of the objects.

polygons — A vector defining polygons which describe the surfaces: it is a concatenation of vectors of the form [m,i_1, ...,i_m] where m is the number of vertices defining a polygon and where *vertices* (*,i_1),...,*vertices*(*,i_m) are those vertices arranged in counter-clockwise order when viewed from outside the object.

parent — (optional) The widget ID of the parent widget.

Output Parameters

shell — (optional) The ID of the newly created widget.

Keywords

position — A two-element vector positioning the widget's upper-left corner (measured in pixels from the upper-left corner of the screen).

shades — A vector specifying the color for each vertex.

size — Two-element vector specifying window size. The default is [500,500].

title — A string specifying the title for the widget.

wid — (output) The window ID of the graphics window.

Examples

```
POLY_SPHERE, 1, 10, 10, v, p

v(1:2,*) = [ 2*v(1,*), 3*v(2,*) ]

WgOrbit, v, p
```

WIN32_PICK_PRINTER Function

Displays a Windows printer dialog.

Usage

printer_name = WIN32_PICK_PRINTER( )

Input Parameters

None.

Returned Value

printername — The name of the printer.

Keywords

None.

See Also

WIN32_PICK_FONT

New PV-WAVE:Database Connection Functions

This section lists the new functions that have been added to PV-WAVE:Database Connection for version 7.5. For complete descriptions, see the *PV-WAVE:Database Connection User's Guide*.

DB_GET_BINARY Function

Returns binary large objects (BLOBS) from a DBMS (database management system) server.

Usage

list_var = DB_GET_BINARY(*handle*, *sql_query*)

Input Parameters

handle — DBMS connection handle (returned by DB_CONNECT).

sql_query — A string containing an SQL statement to execute on the DBMS server. It must be a query (SELECT) statement.

Returned Value

list_var — A PV-WAVE LIST variable, one for each row in the query. Each element in the LIST is a PV-WAVE array of type BYTE.

Keywords

None.

Discussion

Since binary large objects (BLOBS) are transmitted from most DBMS systems in a different way from other data types, using DB_SQL to handle BLOBS would compromise performance.

For queries that return more than one row, specify the order of the rows with the ORDER BY clause in the *sql_query*.

NOTE One column will cause an error.

CAUTION The value of *sql_query* is subject to the following restrictions:

- ❑ It must be a query. UPDATE, INSERT, and/or DELETE will cause an error.
- ❑ It must only return one column. Queries that return more than one column will cause an error.

NULL_PROCESSOR Function

Facilitates the use of the *Null_Info* keyword for the DB_SQL function by extracting the list of rows containing missing data for one or more columns.

Usage

table =
NULL_PROCESSOR(*null_info_object*,*[‘col1’,’col2’,…,’coln’]*,*Comp*=*comp*)

Input Parameters

null_info_object — The object returned by the *Null_Info* keyword in the DB_SQL call.

col_i — The list of column names.

Keywords

Comp=*comp* — Produces the complement to the result, that is, the result contains a list of rows with missing data. *comp* contains a list of rows with no missing data.

Discussion

Assuming the following use of the DB_SQL *Null_Info* keyword:

```
table=db_sql(db_connect(’oracle’, ’user_id/user_pw’), ’select *
   from blanktest’, null_info=foo)
```

where `blanktest` contains the data given below, which has missing data for ID_NO in the 4th, 9th, and 11th rows and missing data for ANIMAL_NAME in the 3rd, 8th, and 10th rows.

ID_NO	ANIMAL_ NAME
1	golden
2	chirpy
3	
	harry
5	KC
6	skip
7	sparky
8	
	sneakers
10	
	harvey

Then,

```
jjj=NULL_PROCESSOR(foo,['ID_NO','ANIMAL_NAME'],Comp=comp)
```

produces the results

```
jjj = 2       3       7       8       9      10
comp = 0           1           4            5            6
```

This output can be utilized as in the following examples.

```
Table2 = table(comp)
```

produces a table with only rows and no missing values or as in the table given above.

ID_NO	ANIMAL_ NAME
1	golden
2	chirpy
5	KC
6	skip
7	sparky

Then,

```
Table3=table(jjj)
```

produces a table containing only rows with missing data (note how zeros have been substituted for values of ID_NO that are missing).

ID_NO	ANIMAL_ NAME
3	
0	harry
8	
0	sneakers
10	
0	harvey

Instead, if you want only the locations where one field is missing, a different db_sql call, jjj=foopro(foo,['ID_NO'],Comp=comp), returns an array, `jjj`, with the rows where ID_NO is missing (`3          8        10`).

Remember that rows are counted beginning with `0`.

New PV-WAVE:IMSL Mathematics Commands

This section lists the new functions and procedures that have been added to PV-WAVE:IMSL Mathematics for version 7.5.

Chapter 4: Quadrature

INTFCN_QMC Function

Integrates a function on a hyper-rectangle using a quasi-Monte Carlo method.

Usage

result = INTFCN_QMC(*f*, *a*, *b*)

Input Parameters

f — Scalar string specifying the user-supplied function to be integrated. Function *f* accepts as input an array of data points at which the function is to be evaluated and returns the scalar value of the function.

a — One-dimensional array containing the lower limits of integration.

b — One-dimensional array containing the upper limits of integration.

Returned Value

The value of

$$\cdot\cdot\int_{a_{n-1}}^{b_{n-1}} f(x_0, \ldots, x_{n-1})\, dx_{n-1}\cdot\cdot$$

is returned. If no value can be computed, then NaN is returned.

Input Keywords

Err_Abs — Absolute accuracy desired.

Default: Err_Abs = 1.e-4.

Err_Rel — Relative accuracy desired.

Default: Err_rel = 1.e-4.

Max_Evals — Number of evaluations allowed.

Default: No limit

Base — The value of *BASE* used to compute the Faure sequence.

Skip — The value of *SKIP* used to compute the Faure sequence.

Double — If present and nonzero, double precision is used.

Output Keywords

Err_est — Named variable into which an estimate of the absolute value of the error is stored.

Discussion

Integration of functions over hypercubes by direct methods, such as INTFCN-HYPER, is practical only for fairly low dimensional hypercubes. This is because the amount of work required increases exponential as the dimension increases.

An alternative to direct methods is Monte Carlo, in which the integral is evaluated as the value of the function averaged over a sequence of randomly chosen points. Under mild assumptions on the function, this method will converge like $1/n^{1/2}$, where n is the number of points at which the function is evaluated.

It is possible to improve on the performance of Monte Carlo by carefully choosing the points at which the function is to be evaluated. Randomly distributed points tend to be non-uniformly distributed. The alternative to at sequence of random points is a *low-discrepancy* sequence. A low-discrepancy sequence is one that is highly uniform.

This function is based on the low-discrepancy Faure sequence, as computed by FAURE_NEXT_PT.

Example

```
FUNCTION F, x
   S = 0.0
   sign = -1.0
   FOR i = 0, N_ELEMENTS(x)-1 DO BEGIN
      prod = 1.0
      FOR j = 0, i DO BEGIN
```

```
            prod = prod*x(j)
         END
         S = S + sign*prod
         sign = -sign
      END
      RETURN, s
   END

   ndim = 10
   a = FLTARR(ndim)
   a(*) = 0
   b = FLTARR(ndim)
   b(*) = 1
   result = intfcn_qmc( 'f', a, b)
   PM, result
      -0.333010
```

Chapter 9: Special Functions

CUM_INTR Function

Evaluates the cumulative interest paid between two periods.

Usage

result = CUM_INTR (*rate*, *n_periods*, *present_value*, *start*, *end_per*, *when*)

Input Parameters

rate — Interest rate.

n_periods — Total number of payment periods. *n_periods* cannot be less than or equal to 0.

present_value — The current value of a stream of future payments, after discounting the payments using some interest rate.

start **—** Starting period in the calculation. *start* cannot be less than 1; or greater than *end_per*.

end_per **—** Ending period in the calculation.

when **—** Time in each period when the payment is made, either 0 for at the end of period or 1 for at the beginning of period.

Returned Value

result **—** The cumulative interest paid between the first period and the last period. If no result can be computed, NaN is returned.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Discussion

Function CUM_INTR evaluates the cumulative interest paid between the first period and the last period.

It is computed using the following:

$$\sum_{i=start}^{end\_per} interest_i$$

where $interest_i$ is computed from the function INT_PAYMENT for the i^{th} period.

Example

In this example, CUM_INTR computes the total interest paid for the first year of a 30-year $200,000 loan with an annual interest rate of 7.25%. The payment is made at the end of each month.

```
PRINT, CUM_INTR(0.0725 / 12, 12*30, 200000., 1, 12, 0)
      -14436.5
```

CUM_PRINC Function

Evaluates the cumulative principal paid between two periods.

Usage

result = CUM_PRINC (*rate, n_periods, present_value, start, end_per, when*)

Input Parameters

rate — Interest rate.

n_periods — Total number of payment periods. *n_periods* cannot be less than or equal to 0.

present_value — The current value of a stream of future payments, after discounting the payments using some interest rate.

start — Starting period in the calculation. *start* cannot be less than 1; or greater than *end_per.*

end_per — Ending period in the calculation.

when — Time in each period when the payment is made, either 0 for at the end of period or 1 for at the beginning of period.

Returned Value

result — The cumulative principal paid between the first period and the last period. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function CUM_PRINC evaluates the cumulative principal paid between the first period and the last period.

It is computed using the following:

$$\sum_{i=start}^{end\_per} principal_i$$

where $principal_i$ is computed from the function PRINC_PAYMENT for the *i*th period.

Example

In this example, CUM_PRINC computes the total principal paid for the first year of a 30-year $200,000 loan with an annual interest rate of 7.25%. The payment is made at the end of each month.

```
PRINT, CUM_PRINC(0.0725 / 12, 12*30, 200000., 1, 12, 0)
      -1935.73
```

DEPRECIATION_DB Function

Evaluates the depreciation of an asset using the fixed-declining balance method.

Usage

result = DEPRECIATION_DB (*cost, salvage, life, period, month*)

Input Parameters

cost — Initial value of the asset.

salvage — The value of an asset at the end of its depreciation period.

life — Number of periods over which the asset is being depreciated.

period — Period for which the depreciation is to be computed. *period* cannot be less than or equal to 0, and cannot be greater than *life* +1.

month — Number of months in the first year. *month* cannot be greater than 12 or less than 1.

Returned Value

result **—** The depreciation of an asset for a specified period using the fixed-declining balance method. If no result can be computed, NaN is returned.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Discussion

Function DEPRECIATION_DB computes the depreciation of an asset for a specified period using the fixed-declining balance method. Function DEPRECIATION_DB varies depending on the specified value for the argument *period*, see table below.

Period	Formula
period = *1*	$cost \times rate \times \frac{month}{12}$
period = *life*	$(cost - total\ depreciation\ from\ periods) \times rate \times \frac{12 - month}{12}$
period other than 1 or *life*	$(cost - total\ depreciation\ from\ prior\ periods) \times rate$

where

$$rate = 1 - \left(\frac{salvage}{cost}\right)^{\left(\frac{1}{life}\right)}$$

NOTE: *rate* is rounded to three decimal places.

Example

In this example, DEPRECIATION_DB computes the depreciation of an asset, which costs $2,500 initially, a useful life of 3 periods and a salvage value of $500, for each period.

```
ans = fltarr(4)
```

```
life = 3
```

```
cost = 2500
salvage = 500
life = 3
month = 6
for period = 1, life+1 DO $
ans(period-1) = depreciation_db(cost, salvage, life, $
                                   period, month)
PM, ans
       518.750
       822.219
       480.998
       140.692
```

DEPRECIATION_DDB Function

Evaluates the depreciation of an asset using the double-declining balance method.

Usage

result = DEPRECIATION_DDB (*cost, salvage, life, period, factor*)

Input Parameters

cost — Initial value of the asset.

salvage — The value of an asset at the end of its depreciation period.

life — Number of periods over which the asset is being depreciated.

period — Period for which the depreciation is to be computed. *period* cannot be greater than *life*.

factor — Rate at which the balance declines. *factor* must be positive.

Returned Value

result — The depreciation of an asset using the double-declining balance method for a period specified by the user. If no result can be computed, NaN is returned.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Discussion

Function DEPRECIATION_DDB computes the depreciation of an asset using the double-declining balance method for a specified period.

It is computed using the following:

$$\left[cost - salvage\left(total\ depreciation\ from\ prior\ periods\right)\right]\left(\frac{factor}{life}\right)$$

Example

In this example, DEPRECIATION_DDB computes the depreciation of an asset, which costs $2,500 initially, lasts 24 periods and a salvage value of $500, for each period.

```
ans = fltarr(24)

life = 24

cost = 2500

salvage = 500

factor = 2

FOR period = 1, life DO $

ans(period-1) = depreciation_ddb(cost, salvage, life, $
                                   period, factor)

PM, ans

       208.333

       190.972

       175.058

       160.470

       147.097

       134.839

       123.603

       113.302

       103.860

       95.2054
```

```
87.2716
79.9990
73.3324
67.2214
61.6196
56.4846
51.7776
47.4628
22.0906
0.00000
0.00000
0.00000
0.00000
0.00000
```

DEPRECIATION_SLN Function

Evaluates the depreciation of an asset using the straight-line method.

Usage

result = DEPRECIATION_SLN (*cost, salvage, life*)

Input Parameters

cost — Initial value of the asset.

salvage — The value of an asset at the end of its depreciation period.

life — Number of periods over which the asset is being depreciated.

Returned Value

result — The straight line depreciation of an asset for its life. If no result can be computed, NaN is returned.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Discussion

Function DEPRECIATION_SLN computes the straight line depreciation of an asset for its life.

It is computed using the following:

$$(cost-salvage)/life$$

Example

In this example, DEPRECIATION_SLN computes the depreciation of an asset, which costs $2,500 initially, lasts 24 periods and a salvage value of $500.

```
PRINT, DEPRECIATION_SLN(2500, 500, 24)
       83.3333
```

DEPRECIATION_SYD Function

Evaluates the depreciation of an asset using the sum-of-years digits method.

Usage

result = DEPRECIATION_SYD (*cost, salvage, life, period*)

Input Parameters

cost **—** Initial value of the asset.

salvage **—** The value of an asset at the end of its depreciation period.

life **—** Number of periods over which the asset is being depreciated.

period **—** Period for which the depreciation is to be computed. *period* cannot be greater than *life*.

Returned Value

result — The sum-of-years digits depreciation of an asset for a specified period. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function DEPRECIATION_SYD computes the sum-of-years digits depreciation of an asset for a specified period.

It is computed using the following:

$$(cost - salvage)(period)\ \frac{(life+1)(life)}{2}$$

Example

In this example, DEPRECIATION_SYD computes the depreciation of an asset, which costs $25,000 initially, lasts 15 years, and a salvage value of $5,000, for the 14th year.

```
PRINT, DEPRECIATION_SYD(25000, 5000, 15, 14)
       333.333
```

DEPRECIATION_VDB Function

Evaluates the depreciation of an asset for any given period using the variable-declining balance method.

Usage

result = DEPRECIATION_VDB (*cost, salvage, life, start, end_per, factor, sln*)

Input Parameters

cost — Initial value of the asset.

salvage — The value of an asset at the end of its depreciation period.

life— Number of periods over which the asset is being depreciated.

start — Starting period in the calculation. *start* cannot be less than 1; or greater than *end_per.*

end_per — Final period for the calculation. *end_per* cannot be greater than *life*.

factor — Rate at which the balance declines. *factor* must be positive.

sln — If equal to zero, do not switch to straight-line depreciation even when the depreciation is greater than the declining balance calculation.

Returned Value

result — The depreciation of an asset for any given period, including partial periods, using the variable-declining balance method. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function DEPRECIATION_VDB computes the depreciation of an asset for any given period using the variable-declining balance method using the following:

If $sln = 0$

$$\sum_{i=start+1}^{end\_per} ddb_i$$

If $sln \neq 0$

$$A + \sum_{i=k}^{end\_per} \frac{cost - A - salvage}{end - k + 1}$$

where ddb_i is computed from the function DEPRECIATION_DDB for the *i*th period. k = the first period where straight-line depreciation is greater than

$$A = \sum_{i=start+1}^{k-1} ddb_i$$

the depreciation using the double-declining balance method.

Example

In this example, DEPRECIATION_VDB computes the depreciation of an asset between the 10th and 15th year, which costs $25,000 initially, lasts 15 years, and has a salvage value of $5,000.

```
PRINT, DEPRECIATION_VDB(25000., 5000., 15, 10, 15, 2, 0)
        976.69
```

DOLLAR_DECIMAL Function

Converts a fractional price to a decimal price.

Usage

result = DOLLAR_DECIMAL (*fractional_num*, *fraction*)

Input Parameters

fractional_num — Whole number of dollars plus the numerator, as the fractional part.

fraction — Denominator of the fractional dollar. *fraction* must be positive.

Returned Value

result — The dollar price expressed as a decimal number. The dollar price is the whole number part of fractional-dollar plus its decimal part divided by fraction. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function DOLLAR_DECIMAL converts a dollar price, expressed as a fraction, into a dollar price, expressed as a decimal number.

It is computed using the following:

$$idollar + \left[fractional\_num - idollar \right] * \frac{10^{(ifrac+1)}}{fraction}$$

where *idollar* is the integer part of *fractional_num*, and *ifrac* is the integer part of *log*(*fraction*).

Example

In this example, DOLLAR_DECIMAL converts $1 1/4 to $1.25.

```
PRINT, DOLLAR_DECIMAL(1.1, 4)
       1.25000
```

DOLLAR_FRACTION Function

Converts a decimal price to a fractional price.

Usage

result = DOLLAR_FRACTION (*decimal_dollar, fraction*)

Input Parameters

decimal_dollar — Dollar price expressed as a decimal number.

fraction — Denominator of the fractional dollar. *fraction* must be positive.

Returned Value

result — The dollar price expressed as a fraction. The numerator is the decimal part of the returned value. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function DOLLAR_FRACTION converts a dollar price, expressed as a decimal number, into a dollar price, expressed as a fractional price. If no result can be computed, NaN is returned.

It can be found by solving the following

$$idollar + \frac{[decimal\_dollar - idollar]}{10^{(ifrac+1)} / fraction}$$

where *idollar* is the integer part of the *decimal_dollar*, and *ifrac* is the integer part of *log*(*fraction*).

Example

In this example, DOLLAR_FRACTION converts $1.25 to $1 1/4.

```
PRINT, DOLLAR_FRACTION(1.25, 4)
          1.10000
```

EFFECTIVE_RATE Function

Evaluates the effective annual interest rate.

Usage

result = EFFECTIVE_RATE (*nominal_rate*, *n_periods*)

Input Parameters

nominal_rate — The interest rate as stated on the face of a security.

n_periods — Number of compounding periods per year.

Returned Value

result — The effective annual interest rate. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function EFFECTIVE_RATE computes the continuously-compounded interest rate equivalent to a given periodically-compounded interest rate. The nominal interest rate is the periodically-compounded interest rate as stated on the face of a security.

It can found by solving the following:

$$\left(1+\frac{nominal\_rate}{n\_periods}\right)^{(n\_periods)}-1$$

Example

In this example, EFFECTIVE_RATE computes the effective annual interest rate of the nominal interest rate, 6%, compounded quarterly.

```
PRINT, EFFECTIVE_RATE(0.06, 4)
     0.0613635
```

FUTURE_VALUE Function

Evaluates the future value of an investment.

Usage

result = FUTURE_VALUE (*rate*, *n_periods*, *payment*, *present_value*, *when*)

Input Parameters

rate — Interest rate.

n_periods— Total number of payment periods.

payment — Payment made in each period.

present_value — The current value of a stream of future payments, after discounting the payments using some interest rate.

when **—** Time in each period when the payment is made, either 0 for at the end of period or 1 for at the beginning of period.

Returned Value

result **—** The future value of an investment. If no result can be computed, NaN is returned.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Discussion

Function FUTURE_VALUE computes the future value of an investment. The future value is the value, at some time in the future, of a current amount and a stream of payments.

It can be found by solving the following:

If $rate = 0$

$$present\_value + (payment)(n\_periods) + future\_value = 0$$

If $rate \neq 0$

$$present\_value(1+rate)^{n\_periods} + payment\left[1+rate(when)\right]\frac{(1+rate)^{n\_periods}-1}{rate}$$
$$+ future\_value = 0$$

Example

In this example, FUTURE_VALUE computes the value of $30,000 payment made annually at the beginning of each year for the next 20 years with an annual interest rate of 5%.

```
PRINT, FUTURE_VALUE(0.05, 20, -30000.00, -30000.00, 1)
   1.12118e+06
```

FUTURE_VAL_SCHD Function

Evaluates the future value of an initial principal taking into consideration a schedule of compound interest rates.

Usage

result = FUTURE_VAL_SCHD (*principal, schedule*)

Input Parameters

principal — Principal or present value.

schedule — One-dimensional array of interest rates to apply.

Returned Value

result — The future value of an initial principal after applying a schedule of compound interest rates. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function FUTURE_VAL_SCHD computes the future value of an initial principal after applying a schedule of compound interest rates.

It is computed using the following with *count* = N_ELEMENTS (*schedule*):

$$\sum_{i=1}^{count}\left(principal * schedule_i\right)$$

where $schedule_i$ = interest rate at the *i*th period.

Example

In this example, FUTURE_VAL_SCHD computes the value of a $10,000 investment after 5 years with interest rates of 5%, 5.1%, 5.2%, 5.3% and 5.4%, respectively.

```
principal = 10000.0
schedule = [ .050, .051, .052, .053, .054 ]
PRINT, FUTURE_VAL_SCHD(principal, schedule)
       12884.8
```

INT_PAYMENT Function

Evaluates the interest payment for an investment for a given period.

Usage

result = INT_PAYMENT (*rate, period, n_periods, present_value, future_value, when*)

Input Parameters

rate — Interest rate.

period— Payment period.

n_periods — Total number of periods.

present_value — The current value of a stream of future payments, after discounting the payments using some interest rate.

future_value — The value, at some time in the future, of a current amount and a stream of payments.

when — Time in each period when the payment is made, either 0 for at the end of period or 1 for at the beginning of period.

Returned Value

result — The interest payment for an investment for a given period. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function INT_PAYMENT computes the interest payment for an investment for a given period.

It is computed using the following:

$$\left\{ present\_value(1+rate)^{n\_periods-1} + payment(1+rate*when)\left[\frac{(1+rate)^{n\_periods-1}}{rate}\right]\right\} rate$$

Example

In this example, INT_PAYMENT computes the interest payment for the second year of a 25-year $100,000 loan with an annual interest rate of 8%. The payment is made at the end of each period.

```
PRINT, INT_PAYMENT(0.08, 2, 25, 100000.00, 0.0, 0)
```

```
      -7890.57
```

INT_RATE_ANNUITY Function

Evaluates the interest rate per period of an annuity.

Usage

result = INT_RATE_ANNUITY (*n_periods, payment, present_value, future_value, when*)

Input Parameters

n_periods — Total number of periods.

payment — Payment made each period.

present_value — The current value of a stream of future payments, after discounting the payments using some interest rate.

future_value — The value, at some time in the future, of a current amount and a stream of payments.

when — Time in each period when the payment is made, either 0 for at the end of period or 1 for at the beginning of period.

Returned Value

result — The interest rate per period of an annuity. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Xguess — If present, the value is used as the initial guess at the interest rate.

Highest — If present, the value is used as the maximum value of the interest rate allowed.

Discussion

Function INT_RATE_ANNUITY computes the interest rate per period of an annuity. An annuity is a security that pays a fixed amount at equally spaced intervals.

It can be found by solving the following:

If $rate = 0$

$$present\_value + (payment)(n\_periods) + future\_value = 0$$

If $rate \neq 0$

$$present\_value(1+rate)^{n\_periods} + payment[1+rate(when)]\frac{(1+rate)^{n\_periods} - 1}{rate}$$
$$+ future\_value = 0$$

Example

In this example, INT_RATE_ANNUITY computes the interest rate of a $20,000 loan that requires 70 payments of $350 to pay off the loan.

```
PRINT, 12*INT_RATE_ANNUITY(70, -350, 20000, 0, 1)
     0.0734513
```

INT_RATE_RETURN Function

Evaluates the internal rate of return for a schedule of cash flows.

Usage

result = INT_RATE_RETURN (*values*)

Input Parameters

values — One-dimensional array of cash flows which occur at regular intervals, which includes the initial investment.

Returned Value

result — The internal rate of return for a schedule of cash flows. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Xguess — If present, the value is used as the initial guess at the internal rate of return.

Highest — If present, the value is used as the maximum value of the internal rate of return allowed.

Discussion

Function INT_RATE_RETURN computes the internal rate of return for a schedule of cash flows. The internal rate of return is the interest rate such that a stream of payments has a net present value of zero.

It is found by solving the following with *count* = N_ELEMENTS (*values*):

$$0 = \sum_{i=1}^{count} \frac{value_i}{(1 + rate)^i}$$

where $value_i$ = the *i*th cash flow, *rate* is the internal rate of return.

Example

In this example, INT_RATE_RETURN computes the internal rate of return for nine cash flows, $-800, $800, $800, $600, $600, $800, $800, $700 and $3,000, with an initial investment of $4,500.

```
values = [ -4500., -800., 800., 800., 600.,  $
              600., 800., 800., 700., 3000. ]
PRINT, INT_RATE_RETURN(values)
    0.0720820
```

INT_RATE_SCHD Function

Evaluates the internal rate of return for a schedule of cash flows. It is not necessary that the cash flows be periodic.

Usage

result = INT_RATE_SCHD (*values*, *dates*)

Input Parameters

values **—** One-dimensional array of cash flows, which includes the initial investment.

dates **—** One-dimensional array of dates cash flows are made. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

Returned Value

result **—** The internal rate of return for a schedule of cash flows that is not necessarily periodic. If no result can be computed, NaN is returned.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Xguess **—** If present, the value is used as the initial guess at the internal rate of return.

Highest **—** If present, the value is used as the maximum value of the internal rate of return allowed.

Discussion

Function INT_RATE_SCHD computes the internal rate of return for a schedule of cash flows that is not necessarily periodic. The internal rate such that the stream of payments has a net present value of zero.

It can be found by solving the following with *count* = N_ELEMENTS (*values*):

$$0 = \sum_{i=1}^{count} \frac{value_i}{(1 + rate)^{\frac{d_i - d_1}{365}}}$$

In the equation above, d_i represents the *i*th payment date. d_1 represents the 1st payment date. $value_i$ represents the *i*th cash flow. *rate* is the internal rate of return.

Example

In this example, INT_RATE_SCHD computes the internal rate of return for nine cash flows, $-800, $800, $800, $600, $600, $800, $800, $700 and $3,000, with an initial investment of $4,500.

```
years = [1998, 1998, 1999, 2000, 2001, 2002, 2003, 2004, $
         2005, 2006]
months = [1, 10, 5, 5, 6, 7, 8, 9, 10, 11]
days = [1, 1, 5, 5, 1, 1, 30, 15, 15, 1]
dates = VAR_TO_DT(years, months, days)
v = [-4500., -800, 800, 800., 600., 600, 800, 800, 700, 3000]
PRINT, INT_RATE_SCHD(v, dates)
     0.0769003
```

MOD_INTERN_RATE Function

Evaluates the modified internal rate of return for a schedule of periodic cash flows.

Usage

result = MOD_INTERN_RATE (v*alues*, *finance_rate*, *reinvest_rate*)

Input Parameters

values — One-dimensional array of cash flows.

finance_rate — Interest paid on the money borrowed.

reinvest_rate — Interest rate received on the cash flows.

Returned Value

result — The modified internal rate of return for a schedule of periodic cash flows. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function MOD_INTERN_RATE computes the modified internal rate of return for a schedule of periodic cash flows. The modified internal rate of return differs from the ordinary internal rate of return in assuming that the cash flows are reinvested at the cost of capital, not at the internal rate of return.

It also eliminates the multiple rates of return problem.

It is computed using the following:

$$\left\{\left[\frac{-(pnpv)(1+reinvest\_rate)^{n\_periods}}{(nnpv)(1+finance\_rate)}\right]^{\frac{1}{n\_periods-1}}\right\}-1$$

where *pnpv* is calculated from the function NET_PRES_VALUE for positive values in *values* using *reinvest_rate*, and where *nnpv* is calculated from the function NET_PRES_VALUE for negative values in *values* using *finance_rate*.

Example

In this example, MOD_INTERN_RATE computes the modified internal rate of return for an investment of $4,500 with cash flows of $-800, $800, $800, $600, $600, $800, $800, $700 and $3,000 for 9 years.

```
value = [ -4500., -800., 800., 800., 600., 600., 800., $
         800., 700., 3000. ]
finance_rate = .08
reinvest_rate = .055
PRINT, MOD_INTERN_RATE(value, finance_rate, reinvest_rate)
     0.0665972
```

NET_PRES_VALUE Function

Evaluates the net present value of a stream of unequal periodic cash flows, which are subject to a given discount rate.

Usage

result = NET_PRES_VALUE (*rate*, *values*)

Input Parameters

rate **—** Interest rate per period.

values **—** One-dimensional array of equally-spaced cash flows.

Returned Value

result **—** The net present value of an investment. If no result can be computed, NaN is returned.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Discussion

Function NET_PRES_VALUE computes the net present value of an investment. Net present value is the current value of a stream of payments, after discounting the payments using some interest rate.

It is found by solving the following with *count* = N_ELEMENTS (*values*):

$$\sum_{i=1}^{count} \frac{value_i}{(1+rate)^i}$$

where $value_i$ = the *i*th cash flow.

Example

In this example, NET_PRES_VALUE computes the net present value of a $10 million prize paid in 20 years ($50,000 per year) with an annual interest rate of 6%.

```
rate = 0.06
value = FLTARR(20)
value(*) = 500000.
PRINT, NET_PRES_VALUE(rate, value)
  5.73496e+06
```

NOMINAL_RATE Function

Evaluates the nominal annual interest rate.

Usage

result = NOMINAL_RATE (*effective_rate*, *n_periods*)

Input Parameters

effective_rate **—** The amount of interest that would be charged if the interest was paid in a single lump sum at the end of the loan.

n_periods **—** Number of compounding periods per year.

Returned Value

result **—** The nominal annual interest rate. If no result can be computed, NaN is returned.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Discussion

Function NOMINAL_RATE computes the nominal annual interest rate. The nominal interest rate is the interest rate as stated on the face of a security.

It is computed using the following:

$$\left[(1+effective\_rate)^{\frac{1}{n\_periods}}-1\right]*n\_periods$$

Example

In this example, NOMINAL_RATE computes the nominal annual interest rate of the effective interest rate, 6.14%, compounded quarterly.

```
PRINT, NOMINAL_RATE(0.0614, 4)
```

```
   0.0600348
```

NUM_PERIODS Function

Evaluates the number of periods for an investment for which periodic and constant payments are made and the interest rate is constant.

Usage

result = NUM_PERIODS (*rate*, *payment*, *present_value*, *future_value*, *when*)

Input Parameters

rate — Interest rate on the investment.

payment — Payment made on the investment.

present_value — The current value of a stream of future payments, after discounting the payments using some interest rate.

future_value — The value, at some time in the future, of a current amount and a stream of payments.

when — Time in each period when the payment is made, either 0 for at the end of period or 1 for at the beginning of period.

Returned Value

result — The number of periods for an investment.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function NUM_PERIODS computes the number of periods for an investment based on periodic, constant payment and a constant interest rate.

It can be found by solving the following:

If $rate = 0$

$$present\_value + (payment)(n\_periods) + future\_value = 0$$

If $rate \neq 0$

$$present\_value(1 + rate)^{n\_periods} + payment\left[1 + rate(when)\right]\frac{(1 + rate)^{n\_periods} - 1}{rate}$$
$$+ future\_value = 0$$

Example

In this example, NUM_PERIODS computes the number of periods needed to pay off a $20,000 loan with a monthly payment of $350 and an annual interest rate of 7.25%. The payment is made at the beginning of each period.

```
PRINT, NUM_PERIODS(0.0725 / 12, -350., 20000., 0., 1)
           70
```

PAYMENT Function

Evaluates the periodic payment for an investment.

Usage

result = PAYMENT (*rate*, *n_periods*, *present_value*, *future_value*, *when*)

Input Parameters

rate — Interest rate.

n_periods — Total number of periods.

present_value — The current value of a stream of future payments, after discounting the payments using some interest rate.

future_value — The value, at some time in the future, of a current amount and a stream of payments.

when — Time in each period when the payment is made, either 0 for at the end of period or 1 for at the beginning of period.

Returned Value

result — The periodic payment for an investment. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function PAYMENT computes the periodic payment for an investment.

It can be found by solving the following:

If $rate = 0$

$$present\_value + (payment)(n\_periods) + future\_value = 0$$

If $rate \neq 0$

$$present\_value(1 + rate)^{n\_periods} + payment\left[1 + rate(when)\right]\frac{(1 + rate)^{n\_periods} - 1}{rate}$$
$$+ future\_value = 0$$

Example

In this example, PAYMENT computes the periodic payment of a 25-year $100,000 loan with an annual interest rate of 8%. The payment is made at the end of each period.

```
PRINT, PAYMENT(0.08, 25, 100000., 0., 0)
      -9367.88
```

PRESENT_VALUE Function

Evaluates the net present value of a stream of equal periodic cash flows, which are subject to a given discount rate..

Usage

result = PRESENT_VALUE (*rate*, *n_periods*, *payment*, *future_value*, *when*)

Input Parameters

rate — Interest rate.

n_periods — Total number of periods.

payment — Payment made in each period.

future_value — The value, at some time in the future, of a current amount and a stream of payments.

when — Time in each period when the payment is made, either 0 for at the end of period or 1 for at the beginning of period.

Returned Value

result — The present value of an investment. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function PRESENT_VALUE computes the present value of an investment.

If $rate = 0$

$$present\_value + (payment)(n\_periods) + future\_value = 0$$

If $rate \neq 0$

$$present\_value(1+rate)^{n\_periods} + payment\left[1+rate(when)\right]\frac{(1+rate)^{n\_periods} - 1}{rate} + future\_value = 0$$

Example

In this example, PRESENT_VALUE computes the present value of 20 payments of $500,000 per payment ($10 million) with an annual interest rate of 6%. The payment is made at the end of each period.

```
PRINT, PRESENT_VALUE(0.06, 20, 500000., 0., 0)
-5.73496e+06
```

PRES_VAL_SCHD Function

Evaluates the present value for a schedule of cash flows. It is not necessary that the cash flows be periodic.

Usage

result = PRES_VAL_SCHD (*rate*, *values*, *dates*)

Input Parameters

rate — Interest rate.

values — One-dimensional array of cash flows.

dates — One-dimensional array of dates cash flows are made. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

Returned Value

result — The present value for a schedule of cash flows that is not necessarily periodic. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function PRES_VAL_SCHD computes the present value for a schedule of cash flows that is not necessarily periodic.

It can be found by solving the following with *count* = N_ELEMENTS (*values*):

$$\sum_{i=1}^{count} \frac{value_i}{(1+rate)^{(d_i-d_1)/365}}$$

In the equation above, d_i represents the *i*th payment date, d_1 represents the 1st payment date, and $value_I$ represents the *i*th cash flow.

Example

In this example, PRES_VAL_SCHD computes the present value of 3 payments, $1,000, $2,000 and $1,000, with an interest rate of 5% made on January 3, 1997, January 3, 1999 and January 3, 2000.

```
rate = 0.05

values = [1000.0, 2000.0, 1000.0]

dates = VAR_TO_DT([1997, 1999, 2000], [1, 1, 1], [3, 3, 3])

PRINT, PRES_VAL_SCHD(rate, values, dates)

       3677.90
```

PRINC_PAYMENT Function

Evaluates the payment on the principal for a specified period.

Usage

result = PRINC_PAYMENT (*rate, period, n_periods, present_value, future_value, when*)

Input Parameters

rate — Interest rate.

period — Payment period.

n_periods — Total number of periods.

present_value — The current value of a stream of future payments, after discounting the payments using some interest rate.

future_value — The value, at some time in the future, of a current amount and a stream of payments.

when — Time in each period when the payment is made, either 0 for at the end of period or 1 for at the beginning of period.

Returned Value

result— The payment on the principal for a given period. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function PRINC_PAYMENT computes the payment on the principal for a given period.

It is computed using the following:

$$payment_i - interest_i$$

where *payment*$_i$ is computed from the function PAYMENT for the *i*th period, *interest*$_i$ is calculated from the function INT_PAYMENT for the *i*th period.

Example

In this example, PRINC_PAYMENT computes the principal paid for the first year on a 30-year $100,000 loan with an annual interest rate of 8%. The payment is made at the end of each year.

```
PRINT, PRINC_PAYMENT(0.08, 1, 30, 100000., 0., 0)

      -882.742
```

ACCR_INT_MAT Function

Evaluates the interest which has accrued on a security that pays interest at maturity.

Usage

result = ACCR_INT_MAT (*issue*, *maturity*, *coupon_rate*, *par_value*, *basis*)

Input Parameters

issue — The date on which interest starts accruing. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

coupon_rate — Annual interest rate set forth on the face of the security; the coupon rate.

par_value — Nominal or face value of the security used to calculate interest payments.

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	PDay count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result **—** The interest which has accrued on a security that pays interest at maturity. If no result can be computed, NaN is returned.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Discussion

Function ACCR_INT_MAT computes the accrued interest for a security that pays interest at maturity:

$$(par\_value)(rate)\left(\frac{A}{D}\right)$$

In the above equation, *A* represents the number of days starting at issue date to maturity date and *D* represents the annual basis.

Example

In this example, ACCR_INT_MAT computes the accrued interest for a security that pays interest at maturity using the US (NASD) 30/360 day count method. The security has a par value of $1,000, the issue date of October 1, 2000, the maturity date of November 3, 2000, and a coupon rate of 6%.

```
issue = VAR_TO_DT(2000, 10, 1)
maturity = VAR_TO_DT(2000, 11, 3)
rate = .06
par = 1000.
basis = 1
PRINT, ACCR_INT_MAT(issue, maturity, rate, par, basis)
       5.33333
```

ACCR_INT_PER Function

Evaluates the interest which has accrued on a security that pays interest periodically.

Usage

result = ACCR_INT_PER (*issue*, *first_coupon*, *settlement*, *coupon_rate*, *par_value*, *frequency*, *basis*)

Input Parameters

issue — The date on which interest starts accruing. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

first_coupon — First date on which an interest payment is due on the security (e.g. coupon date). For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

coupon_rate — Annual interest rate set forth on the face of the security; the coupon rate.

par_value — Nominal or face value of the security used to calculate interest payments.

frequency — Frequency of the interest payments. It should be either 1, 2 or 4.

frequency	Meaning
1	One payment per year (Annual payment)
2	Two payments per year (Semi-annual payment)
4	Four payments per year (Quarterly payment)

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	PDay count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The accrued interest for a security that pays periodic interest. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function ACCR_INT_PER computes the accrued interest for a security that pays periodic interest.

In the equation below, A_i represents the number days which have accrued for the *i*th quasi-coupon period within the odd period. (The quasi-coupon periods are periods obtained by extending the series of equal payment periods to before or after the actual payment periods.) *NC* represents the number of quasi-coupon periods within the odd period, rounded to the next highest integer. (The odd period is a period between payments that differs from the usual equally spaced periods at which payments are made.) NL_i represents the length of the normal ith quasi-coupon period within the odd period. NL_I is expressed in days.

Function ACCR_INT_PER can be found by solving the following:

$$(par\_value)\left(\frac{rate}{frequency}\left[\sum_{i=1}^{NC}\left(\frac{A_i}{NL_i}\right)\right]\right)$$

Example

In this example, ACCR_INT_PER computes the accrued interest for a security that pays periodic interest using the US (NASD) 30/360 day count method. The security has a par value of $1,000, the issue date of October 1, 1999, the settlement date of November 3, 1999, the first coupon date of March 31, 2000, and a coupon rate of 6%.

```
issue = VAR_TO_DT(1999, 10, 1)

first_coupon = VAR_TO_DT(2000, 3, 31)

settlement = VAR_TO_DT(1999, 11, 3)

rate = .06

par = 1000.

frequency = 2

basis = 1

PRINT, ACCR_INT_PER(issue, first_coupon, settlement, $
                     rate, par, frequency, basis)

       5.33333
```

BOND_EQV_YIELD Function

Evaluates the bond-equivalent yield of a Treasury bill.

Usage

result = BOND_EQV_YIELD (*settlement*, *maturity*, *discount_rate*)

Input Parameters

settlement **—** The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity **—** The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

discount_rate **—** The interest rate implied when a security is sold for less than its value at maturity in lieu of interest payments.

Returned Value

result **—** The bond-equivalent yield of a Treasury bill. If no result can be computed, NaN is returned.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Discussion

Function BOND_EQV_YIELD computes the bond-equivalent yield for a Treasury bill.

It is computed using the following:

if DSM <=182

$$\frac{365 * discount\_rate}{360 - discount\_rate * DSM}$$

otherwise,

$$\frac{-\frac{DSM}{365} + \sqrt{\left(\frac{DSM}{365}\right)^2 - \left(2 * \frac{DSM}{365} - 1\right) * \frac{discount\_rate * DSM}{discount\_rate * DSM - 360}}}{\frac{DSM}{365} - 0.5}$$

In the above equation, *DSM* represents the number of days starting at settlement date to maturity date.

Example

In this example, BOND_EQV_YIELD computes the bond-equivalent yield for a Treasury bill with the settlement date of July 1, 1999, the maturity date of July 1, 2000, and discount rate of 5% at the issue date.

```
PRINT, BOND_EQV_YIELD(settlement, maturity, discount)
     0.052857
```

CONVEXITY Function

Evaluates the convexity for a security.

Usage

result = CONVEXITY (*settlement, maturity, coupon_rate, yield, frequency, basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

coupon_rate — Annual interest rate set forth on the face of the security; the coupon rate.

yield — Annual yield of the security.

frequency — Frequency of the interest payments. It should be either 1, 2 or 4.

frequency	**Meaning**
1	One payment per year (Annual payment)
2	Two payments per year (Semi-annual payment)
4	Four payments per year (Quarterly payment)

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	**Day count basis**
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The convexity for a security. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function CONVEXITY computes the convexity for a security. Convexity is the sensitivity of the duration of a security to changes in yield.

It is computed using the following:

$$\frac{\dfrac{1}{(q* \, frequency)^2}\left\{\sum_{t=1}^{n} t(t+1)\left(\dfrac{coupon\_rate}{frequency}\right)q^{-t} + n(n+1)q^{-n}\right\}}{\left(\sum_{t=1}^{n}\left(\dfrac{coupon\_rate}{frequency}\right)q^{-t} + q^{-n}\right)}$$

where n is calculated from the function COUPON_NUM and

$$q = 1 + \frac{yield}{frequency}.$$

Example

In this example, CONVEXITY computes the convexity for a security with the settlement date of July 1, 1990, and maturity date of July 1, 2000, using the Actual/365 day count method.

```
settlement = VAR_TO_DT(1990, 7, 1)

maturity = VAR_TO_DT(2000, 7, 1)

coupon = .075

yield = .09

frequency = 2

basis = 3

PRINT, CONVEXITY(settlement, maturity, $
                  coupon, yield, frequency, basis)

       59.4050
```

COUPON_DAYS Function

Evaluates the number of days in the coupon period containing the settlement date.

Usage

result = COUPON_DAYS (*settlement*, *maturity*, *frequency*, *basis*)

Input Parameters

settlement **—** The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity **—** The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

frequency **—** Frequency of the interest payments. It should be either 1, 2 or 4.

frequency	Meaning
1	One payment per year (Annual payment)
2	Two payments per year (Semi-annual payment)
4	Four payments per year (Quarterly payment)

basis **—** The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result **—** The number of days in the coupon period which contains the settlement date. If no result can be computed, NaN is returned.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Discussion

Function COUPON_DAYS computes the number of days in the coupon period that contains the settlement date. For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol. 1, pages 17-35.

Example

In this example, COUPON_DAYS computes the number of days in the coupon period of a bond with the settlement date of November 11, 1996, and the maturity date of March 1, 2009, using the Actual/365 day count method.

```
settlement = VAR_TO_DT(1996, 11, 11)
maturity = VAR_TO_DT(2009, 3, 1)
frequency =  2
basis =  3
PRINT, COUPON_DAYS(settlement, maturity, frequency, basis)
       182.500
```

COUPON_NUM Function

Evaluates the number of coupons payable between the settlement date and the maturity date.

Usage

result = COUPON_NUM (*settlement, maturity, frequency, basis*)

Input Parameters

settlement **—** The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity **—** The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

frequency **—** Frequency of the interest payments. It should be either 1, 2 or 4.

basis **—** The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result **—** The number of coupons payable between the settlement date and the maturity date.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Discussion

Function COUPON_NUM computes the number of coupons payable between the settlement date and the maturity date. For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol. 1, pages 17-35.

Example

In this example, COUPON_NUM computes the number of coupons payable with the settlement date of November 11, 1996, and the maturity date of March 1, 2009, using the Actual/365 day count method.

```
settlement = VAR_TO_DT(1996, 11, 11)

maturity = VAR_TO_DT(2009, 3, 1)

frequency = 2

basis = 3
```

```
PRINT, COUPON_NUM(settlement, maturity, frequency, basis)
          25
```

SETTLEMENT_DB Function

Evaluates the number of days starting with the beginning of the coupon period and ending with the settlement date.

Usage

result = SETTLEMENT_DB (*settlement, maturity, frequency, basis*)

Input Parameters

settlement **—** The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity **—** The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

frequency **—** Frequency of the interest payments. It should be either 1, 2 or 4.

frequency	Meaning
1	One payment per year (Annual payment)
2	Two payments per year (Semi-annual payment)
4	Four payments per year (Quarterly payment)

basis **—** The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual

basis	Day count basis
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result **—** The number of days in the period starting with the beginning of the coupon period and ending with the settlement date.

Input Keywords

double **—** If present and nonzero, double precision is used.

Discussion

Function SETTLEMENT_DB computes the number of days from the beginning of the coupon period to the settlement date. For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol. 1, pages 17-35.

Example

In this example, SETTLEMENT_DB computes the number of days from the beginning of the coupon period to November 11, 1996, of a bond with the maturity date of March 1, 2009, using the Actual/365 day count method.

```
settlement = VAR_TO_DT(1996, 11, 11)
maturity = VAR_TO_DT(2009, 3, 1)
frequency = 2
basis = 3
PRINT, SETTLEMENT_DB(settlement, maturity, frequency, basis)
            71
```

COUPON_DNC Function

Evaluates the number of days starting with the settlement date and ending with the next coupon date.

Usage

result = COUPON_DNC (*settlement, maturity, frequency, basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

frequency — Frequency of the interest payments. It should be either 1, 2 or 4.

frequency	Meaning
1	One payment per year (Annual payment)
2	Two payments per year (Semi-annual payment)
4	Four payments per year (Quarterly payment)

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360

basis	Day count basis
3	Actual/365
4	European 30/360

Returned Value

result **—** The number of days starting with the settlement date and ending with the next coupon date.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Discussion

Function COUPON_DNC computes the number of days from the settlement date to the next coupon date. For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol. 1, pp. 17-35.

Example

In this example, COUPON_DNC computes the number of days from November 11, 1996, to the next coupon date of a bond with the maturity date of March 1, 2009, using the Actual/365 day count method.

```
settlement =  VAR_TO_DT(1996, 11, 11)

maturity = VAR_TO_DT(2009, 3, 1)

frequency = 2

basis = 3

PRINT, COUPON_DNC(settlement, maturity, frequency, basis)

           110
```

DEPREC_AMORDEGRC Function

Evaluates the depreciation for each accounting period. During the evaluation of the function a depreciation coefficient based on the asset life is applied.

Usage

result = DEPREC_AMORDEGRC (*cost*, *issue*, *first_period*, *salvage*, *period*, *rate*, *basis*)

Input Parameters

cost — Initial value of the asset.

issue — The date on which interest starts accruing. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

first_period — Date of the end of the first period. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

salvage — The value of an asset at the end of its depreciation period.

period — Depreciation for the accounting period to be computed.

rate — Depreciation rate.

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result **—** The depreciation for each accounting period. If no result can be computed, NaN is returned.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Discussion

Function DEPREC_AMORDEGRC computes the depreciation for each accounting period. This function is similar to DEPREC_AMORLINC. However, in this function a depreciation coefficient based on the asset life is applied during the evaluation of the function.

Example

In this example, DEPREC_AMORDEGRC computes the depreciation for the second accounting period using the US (NASD) 30/360 day count method. The security has the issue date of November 1, 1999, end of first period of November 30, 2000, cost of $2,400, salvage value of $300, depreciation rate of 15%.

```
issue = VAR_TO_DT(1999, 11, 1)

first_period = VAR_TO_DT(2000, 11, 30)

cost = 2400.

salvage = 300.

period = 2

rate = .15

basis = 1

PRINT, DEPREC_AMORDEGRC(cost, issue, first_period, $
                         salvage, period, rate, basis

       335.000
```

DEPREC_AMORLINC Function

Evaluates the depreciation for each accounting period. This function is similar to DEPREC_AMORDEGRC, except that DEPREC_AMORDEGRC has a depreciation coefficient that is applied during the evaluation that is based on the asset life.

Usage

result = DEPREC_AMORLINC (*cost*, *issue, first_period*, *salvage, period, rate*, *basis*)

Input Parameters

cost — Initial value of the asset.

issue — The date on which interest starts accruing. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

first_period— Date of the end of the first period. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

salavge — The value of an asset at the end of its depreciation period.

period— Depreciation for the accounting period to be computed.

rate — Depreciation rate.

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The depreciation for each accounting period. If no result can be computed, NaN is returned.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Discussion

Function DEPREC_AMORLINC computes the depreciation for each accounting period.

Example

In this example, DEPREC_AMORLINC computes the depreciation for the second accounting period using the US (NASD) 30/360 day count method. The security has the issue date of November 1, 1999, end of first period of November 30, 2000, cost of $2,400, salvage value of $300, depreciation rate of 15%.

```
issue = VAR_TO_DT(1999, 11, 1)

first_period = VAR_TO_DT(2000, 11, 30)

cost = 2400.

salvage = 300.

period = 2

rate = .15

basis = 1

PRINT, DEPREC_AMORLINC(cost, issue, first_period, $
                       salvage, period, rate, basis)

       360.000
```

DISCOUNT_PR Function

Evaluates the price of a security sold for less than its face value.

Usage

result = DISCOUNT_PR (*settlement*, *maturity*, *discount_rate*, *redemption*, *basis*)

Input Parameters

settlement **—** The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity **—** The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

discount_rate **—** The interest rate implied when a security is sold for less than its value at maturity in lieu of interest payments.

redemption **—** Redemption value per $100 face value of the security.

basis **—** The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	**Day count basis**
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result **—** The price per face value for a discounted security. If no result can be computed, NaN is returned.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Discussion

Function DISCOUNT_PR computes the price per $100 face value of a discounted security.

It is computed using the following:

$$redemption - (discount\_rate)\left[redemption\left(\frac{DSM}{B}\right)\right]$$

In the equation above, *DSM* represents the number of days starting at the settlement date and ending with the maturity date. *B* represents the number of days in a year based on the annual basis.

Example

In this example, DISCOUNT_PR computes the price of the discounted bond with the settlement date of July 1, 2000, and maturity date of July 1, 2001, at the discount rate of 5% using the US (NASD) 30/360 day count method.

```
settlement = VAR_TO_DT(2000, 7, 1)

maturity = VAR_TO_DT(2001, 7, 1)

discount = .05

redemption = 100.

basis = 1

PRINT, DISCOUNT_PR(settlement, maturity, discount, $
                   redemption, basis

       95.0000
```

DISCOUNT_RT Function

Evaluates the interest rate implied when a security is sold for less than its value at maturity in lieu of interest payments.

Usage

result = DISCOUNT_RT (*settlement, maturity, price, redemption, basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity **—** The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

price **—** Price per $100 face value of the security.

redemption **—** Redemption value per $100 face value of the security.

basis **—** The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result **—** The discount rate for a security. If no result can be computed, NaN is returned.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Discussion

Function DISCOUNT_RT computes the discount rate for a security. The discount rate is the interest rate implied when a security is sold for less than its value at maturity in lieu of interest payments.

It is computed using the following:

$$\left(\frac{redemption - price}{price}\right)\left(\frac{B}{DSM}\right)$$

In the equation above, *B* represents the number of days in a year based on the annual basis and *DSM* represents the number of days starting with the settlement date and ending with the maturity date.

Example

In this example, DISCOUNT_RT computes the discount rate of a security which is selling at $97.975 with the settlement date of February 15, 2000, and maturity date of June 10, 2000, using the Actual/365 day count method.

```
settlement = VAR_TO_DT(2000, 2, 15)

maturity = VAR_TO_DT(2000, 6, 10)

price = 97.975

redemption = 100.

basis = 3

PRINT, DISCOUNT_RT(settlement, maturity, price,  $
                   redemption, basis)

     0.0637177
```

DISCOUNT_YLD Function

Evaluates the annual yield of a discounted security.

Usage

result = DISCOUNT_YLD (*settlement*, *maturity*, *price*, *redemption*, *basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

price — Price per $100 face value of the security.

redemption — Redemption value per $100 face value of the security.

basis **—** The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result **—** The annual yield for a discounted security. If no result can be computed, NaN is returned.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Discussion

Function DISCOUNT_YLD computes the annual yield for a discounted security.

It is computed using the following:

$$\left(\frac{redemption - price}{price}\right)\left(\frac{B}{DSM}\right)$$

In the equation above, *B* represents the number of days in a year based on the annual basis, and *DSM* represents the number of days starting with the settlement date and ending with the maturity date.

Example

In this example, DISCOUNT_YLD computes the annual yield for a discounted security which is selling at $95.40663 with the settlement date of July 1, 1995, and maturity date of July 1, 2005, using the US (NASD) 30/360 day count method.

```
settlement = VAR_TO_DT(1995, 7, 1)

maturity = VAR_TO_DT(2005, 7, 1)

price = 95.40663

redemption = 105.

basis = 1

PRINT, DISCOUNT_YLD(settlement, maturity, price, redemption$
                      basis)

0.0100552
```

DURATION Function

Evaluates the annual duration of a security where the security has periodic interest payments.

Usage

result = DURATION (*settlement*, *maturity*, *coupon_rate*, *yield*, *frequency*, *basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

coupon_rate — Annual interest rate set forth on the face of the security; the coupon rate.

yield — Annual yield of the security.

frequency— Frequency of the interest payments. It should be either 1, 2 or 4.

frequency	Meaning
1	One payment per year (Annual payment)
2	Two payments per year (Semi-annual payment)
4	Four payments per year (Quarterly payment)

basis **—** The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result **—** The annual duration of a security with periodic interest payments. If no result can be computed, NaN is returned.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Discussion

Function DURATION computes the Maccaluey's duration of a security with periodic interest payments. The Maccaluey's duration is the weighted-average time to the payments, where the weights are the present value of the payments.

It is computed using the following:

$$\left(\frac{\dfrac{\frac{DSC}{E}*100}{\left(1+\frac{yield}{freq}\right)^{\left(N-1+\frac{DSC}{E}\right)}} + \sum_{k=1}^{N}\left(\left(\dfrac{100*coupon\_rate}{freq*\left(1+\frac{yield}{freq}\right)^{\left(k-1+\frac{DSC}{E}\right)}}\right)*\left(k-1+\frac{DSC}{E}\right)\right)}{\dfrac{100}{\left(1+\frac{yield}{freq}\right)^{N-1+\frac{DSC}{E}}} + \sum_{k=1}^{N}\left(\dfrac{100*coupon\_rate}{freq*\left(1+\frac{yield}{freq}\right)^{k-1+\frac{DSC}{E}}}\right)} \right) * \frac{1}{freq}$$

In the equation above, *DSC* represents the number of days starting with the settlement date and ending with the next coupon date. *E* represents the number of days within the coupon period. *N* represents the number of coupons payable from the settlement date to the maturity date. *freq* represents the frequency of the coupon payments annually.

Example

In this example, DURATION computes the annual duration of a security with the settlement date of July 1, 1995, and maturity date of July 1, 2005, using the Actual/365 day count method.

```
settlement = VAR_TO_DT(1995, 7, 1)

maturity = VAR_TO_DT(2005, 7, 1)

coupon = .075

yield = .09

frequency = 2

basis = 3

PRINT, DURATION(settlement, maturity, coupon,  $
                 yield, frequency, basis)

       7.04195
```

INT_RATE_SEC Function

Evaluates the interest rate of a fully invested security.

Usage

result = INT_RATE_SEC (*settlement*, *maturity*, *investment*, *redemption*, *basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

investment — The total amount one has invested in the security.

redemption — Amount to be received at maturity.

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The interest rate for a fully invested security. If no result can be computed, NaN is returned.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Discussion

Function INT_RATE_SEC computes the interest rate for a fully invested security.

It is computed using the following:

$$\left(\frac{redemption - investment}{investment}\right)\left(\frac{B}{DSM}\right)$$

In the equation above, *B* represents the number of days in a year based on the annual basis, and *DSM* represents the number of days in the period starting with the settlement date and ending with the maturity date.

Example

In this example, INT_RATE_SEC computes the interest rate of a $7,000 investment with the settlement date of July 1, 1995, and maturity date of July 1, 2005, using the Actual/365 day count method. The total amount received at the end of the investment is $10,000.

```
settlement = VAR_TO_DT(1995, 7, 1)

maturity = VAR_TO_DT(2005, 7, 1)

investment = 7000.

redemption = 10000.

basis = 3

PRINT, INT_RATE_SEC(settlement, maturity, investment,$
                    redemption, basis)

     0.0428219
```

DURATION_MAC Function

Evaluates the modified Macauley duration of a security.

Usage

result = DURATION_MAC (*settlement*, *maturity*, *coupon_rate*, *yield*, *frequency*, *basis*)

Input Parameters

settlement **—** The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity **—** The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

coupon_rate **—** Annual interest rate set forth on the face of the security; the coupon rate.

yield **—** Annual yield of the security.

frequency **—** Frequency of the interest payments. It should be either 1, 2 or 4.

frequency	Meaning
1	One payment per year (Annual payment)
2	Two payments per year (Semi-annual payment)
4	Four payments per year (Quarterly payment)

basis **—** The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual

basis	Day count basis
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

***result* —** The modified Macauley duration of a security is returned. The security has an assumed par value of $100. If no result can be computed, NaN is returned.

Input Keywords

***Double* —** If present and nonzero, double precision is used.

Discussion

Function DURATION_MAC computes the modified Macauley duration for a security with an assumed par value of $100.

It is computed using the following:

$$\frac{duration}{1 \quad \frac{yield}{frequency}}$$

where *duration* is calculated from the function DURATION.

Example

In this example, DURATION_MAC computes the modified Macauley duration of a security with the settlement date of July 1, 1995, and maturity date of July 1, 2005, using the Actual/365 day count method.

```
settlement = VAR_TO_DT(1995, 7, 1)
maturity = VAR_TO_DT(2005, 7, 1)
coupon = .075
yield = .09
```

```
frequency = 2
basis = 3
PRINT, DURATION_MAC(settlement, maturity, $
                    coupon, yield, frequency, basis)
      6.73871
```

COUPON_NCD Function

Evaluates the first coupon date which follows the settlement date.

Usage

result = COUPON_NCD (*settlement, maturity, frequency, basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

frequency — Frequency of the interest payments. It should be either 1, 2 or 4.

frequency	Meaning
1	One payment per year (Annual payment)
2	Two payments per year (Semi-annual payment)
4	Four payments per year (Quarterly payment)

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result **—** The first coupon date which follows the settlement date.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Discussion

Function COUPON_NCD computes the next coupon date after the settlement date. For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol 1, pages 17-35.

Example

In this example, COUPON_NCD computes the next coupon date of a bond with the settlement date of November 11, 1996, and the maturity date of March 1, 2009, using the Actual/365 day count method.

```
settlement = VAR_TO_DT(1996, 11, 11)
maturity = VAR_TO_DT(2009, 3, 1)
frequency = 2
basis = 3
ans = COUPON_NCD(settlement, maturity, frequency, basis)
DT_TO_STR, ans, d, Date_Fmt=4
01/March/1997
```

COUPON_PCD Function

Evaluates the coupon date which immediately precedes the settlement date.

Usage

result = COUPON_PCD (*settlement*, *maturity*, *frequency*, *basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

fequency — Frequency of the interest payments. It should be either 1, 2 or 4.

frequency	Meaning
1	One payment per year (Annual payment)
2	Two payments per year (Semi-annual payment)
4	Four payments per year (Quarterly payment)

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365

basis	Day count basis
4	European 30/360

Returned Value

result **—** The coupon date which immediately precedes the settlement date.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Discussion

Function COUPON_PCD computes the coupon date which immediately precedes the settlement date. For a good discussion on day count basis, see *SIA Standard Securities Calculation Methods* 1993, vol 1, pages 17-35.

Example

In this example, COUPON_PCD computes the previous coupon date of a bond with the settlement date of November 11, 1986, and the maturity date of March 1, 1999, using the Actual/365 day count method.

```
settlement = VAR_TO_DT(1996, 11, 11)
maturity = VAR_TO_DT(2009, 3, 1)
frequency = 2
basis = 3
ans = COUPON_PCD(settlement, maturity, frequency, basis)
DT_TO_STR, ans, d, Date_Fmt=4
PRINT, d
01/September/1996
```

PRICE_PERIODIC Function

Evaluates the price, per $100 face value, of a security that pays periodic interest.

Usage

result = PRICE_PERIODIC (*settlement, maturity, rate, yield, redemption, frequency, basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid.

rate — Annual interest rate set forth on the face of the security; the coupon rate.

yield — Annual yield of the security.

redemption — Redemption value per $100 face value of the security.

frequency — Frequency of the interest payments. It should be either 1, 2 or 4.

frequency	**Meaning**
1	One payment per year (Annual payment)
2	Two payments per year (Semi-annual payment)
4	Four payments per year (Quarterly payment)

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result **—** The price per $100 face value of a security that pays periodic interest. If no result can be computed, NaN is returned.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Discussion

Function PRICE_PERIODIC computes the price per $100 face value of a security that pays periodic interest.

It is computed using the following:

$$\left(\frac{redemption}{\left(1+\frac{yield}{frequency}\right)^{\left(N-1+\frac{DSC}{E}\right)}}\right)+\left[\sum_{k=1}^{N}\frac{100*\frac{rate}{frequency}}{\left(1+\frac{yield}{frequency}\right)^{\left(k-1+\frac{DSC}{E}\right)}}\right]-\left(100*\frac{rate}{frequency}*\frac{A}{E}\right)$$

In the above equation, *DSC* represents the number of days in the period starting with the settlement date and ending with the next coupon date. *E* represents the number of days within the coupon period. *N* represents the number of coupons payable in the timeframe from the settlement date to the redemption date. *A* represents the number of days in the timeframe starting with the beginning of coupon period and ending with the settlement date.

Example

In this example, PRICE_PERIODIC computes the price of a bond that pays coupon every six months with the settlement of July 1, 1995, the maturity date of July 1, 2005, a annual rate of 6%, annual yield of 7% and redemption value of $105 using the US (NASD) 30/360 day count method.

```
settlement = VAR_TO_DT(1995, 7, 1)

maturity = VAR_TO_DT(2005, 7, 1)

rate = .06

yield = .07

redemption = 105.

frequency = 2

basis = 1

PRINT, PRICE_PERIODIC(settlement, maturity, rate, yield, $
                       redemption, frequency, basis)

       95.4067
```

PRICE_MATURITY Function

Evaluates the price, per $100 face value, of a security that pays interest at maturity.

Usage

result = PRICE_MATURITY (*settlement*, *maturity*, *issue*, *rate*, *yield*, *basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

issue — The date on which interest starts accruing. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

rate — Annual interest rate set forth on the face of the security; the coupon rate.

yield — Annual yield of the security.

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	**Day count basis**
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The price per $100 face value of a security that pays interest at maturity. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function PRICE_MATURITY computes the price per $100 face value of a security that pays interest at maturity.

It is computed using the following:

$$\left[\frac{100+\left(\frac{DIM}{B}*rate*100\right)}{1+\left(\frac{DSM}{B}*yield\right)}\right]-\left(\frac{A}{B}*rate*100\right)$$

In the equation above, *B* represents the number of days in a year based on the annual basis. *DSM* represents the number of days in the period starting with the settlement date and ending with the maturity date. *DIM* represents the number of days in the period starting with the issue date and ending with the maturity date. *A* represents the number of days in the period starting with the issue date and ending with the settlement date.

Example

In this example, PRICE_MATURITY computes the price at maturity of a security with the settlement date of August 1, 2000, maturity date of July 1, 2001 and issue date of July 1, 2000, using the US (NASD) 30/360 day count method. The security has 5% annual yield and 5% interest rate at the date of issue.

```
settlement = VAR_TO_DT(2000, 8, 1)

maturity = VAR_TO_DT(2001, 7, 1)

issue = VAR_TO_DT(2000, 7, 1)

rate = .05

yield = .05

basis = 1

PRINT, PRICE_MATURITY(settlement, maturity, issue, $
                       rate, yield, basis)

       99.9817
```

MATURITY_REC Function

Evaluates the amount one receives when a fully invested security reaches the maturity date.

Usage

result = MATURITY_REC (*settlement, maturity, investment, discount_rate, basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User’s Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User’s Guide.

investment — The total amount one has invested in the security.

discount_rate — The interest rate implied when a security is sold for less than its value at maturity in lieu of interest payments.

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The amount one receives when a fully invested security reaches its maturity date. If no result can be computed, NaN is returned.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Function MATURITY_REC computes the amount received at maturity for a fully invested security.

It is computed using the following:

$$\frac{investment}{1-\left(discount\_rate*\frac{DIM}{B}\right)}$$

In the equation above, *B* represents the number of days in a year based on the annual basis, and *DIM* represents the number of days in the period starting with the issue date and ending with the maturity date.

Example

In this example, MATURITY_REC computes the amount received of a $7,000 investment with the settlement date of July 1, 1995, maturity date of July 1, 2005 and discount rate of 6%, using the Actual/365 day count method.

```
settlement = VAR_TO_DT(1995, 7, 1)

maturity = VAR_TO_DT(2005, 7, 1)

investment = 7000.

discount = .06

basis =  3

PRINT, MATURITY_REC(settlement, maturity, investment,$
                    discount, basis)

       17521.6
```

TBILL_PRICE Function

Evaluates the price per $100 face value of a Treasury bill.

Usage

result = TBILL_PRICE (*settlement*, *maturity*, *discount_rate*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

discount_rate — The interest rate implied when a security is sold for less than its value at maturity in lieu of interest payments.

Returned Value

result — The price per $100 face value of a Treasury bill. If no result can be computed, NaN is returned.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Discussion

Function TBILL_PRICE computes the price per $100 face value for a Treasury bill.

It is computed using the following:

$$100\left(1-\frac{discount\_rate * DSM}{360}\right)$$

In the equation above, *DSM* represents the number of days in the period starting with the settlement date and ending with the maturity date (any maturity date that is more than one calendar year after the settlement date is excluded).

Example

In this example, TBILL_PRICE computes the price for a Treasury bill with the settlement date of July 1, 2000, the maturity date of July 1, 2001, and a discount rate of 5% at the issue date.

```
settlement = VAR_TO_DT(2000, 7, 1)
maturity = VAR_TO_DT(2001, 7, 1)
discount = .05
PRINT, TBILL_PRICE(settlement, maturity, discount)
       94.9306
```

TBILL_YIELD Function

Evaluates the yield of a Treasury bill.

Usage

result = TBILL_YIELD (*settlement*, *maturity*, *price*)

Input Parameters

***settlement* —** The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

***maturity* —** The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

***price* —** Price per $100 face value of the Treasury bill.

Returned Value

***result* —** The yield for a Treasury bill. If no result can be computed, NaN is returned.

Input Keywords

***Double* —** If present and nonzero, double precision is used.

Discussion

Function TBILL_YIELD computes the yield for a Treasury bill.

It is computed using the following:

$$\left(\frac{100 - price}{price}\right)\left(\frac{360}{DSM}\right)$$

In the equation above, *DSM* represents the number of days in the period starting with the settlement date and ending with the maturity date (any maturity date that is more than one calendar year after the settlement date is excluded).

Example

In this example, TBILL_YIELD computes the yield for a Treasury bill with the settlement date of July 1, 2000, the maturity date of July 1, 2001, and priced at $94.93.

```
settlement = VAR_TO_DT(2000, 7, 1)

maturity = VAR_TO_DT(2001, 7, 1)

price = 94.93
```

```
PRINT, TBILL_YIELD(settlement, maturity, price)
     0.0526762
```

YEAR_FRACTION Function

Evaluates the fraction of a year represented by the number of whole days between two dates.

Usage

result = YEAR_FRACTION (*date_start*, *date_end*, *basis*)

Input Parameters

date_start — Initial date. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

date_end — Ending date. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result — The fraction of a year represented by the number of whole days between two dates. If no result can be computed, NaN is returned.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Discussion

Function YEAR_FRACTION computes the fraction of the year.

It is computed using the following:

$$A / D$$

where A = the number of days from *start* to *end*, D = annual basis.

Example

In this example, YEAR_FRACTION computes the year fraction between August 1, 2000, and July 1, 2001, using the US (NASD) 30/360 day count method.

```
date_start = VAR_TO_DT(2000, 8, 1)

date_end = VAR_TO_DT(2001, 7, 1)

basis = 1

PRINT, YEAR_FRACTION(date_start, date_end, basis)

      0.916667
```

YIELD_MATURITY Function

Evaluates the annual yield of a security that pays interest at maturity.

Usage

result = YIELD_MATURITY (*settlement, maturity, issue, rate, price, basis*)

Input Parameters

settlement **—** The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User’s Guide.

maturity **—** The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

issue **—** The date on which interest starts accruing. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

rate **—** Interest rate at date of issue of the security.

price **—** Price per $100 face value of the security.

basis **—** The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result **—** The annual yield of a security that pays interest at maturity. If no result can be computed, NaN is returned.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Discussion

Function YIELD_MATURITY computes the annual yield of a security that pays interest at maturity.

It is computed using the following:

$$\left\{\frac{\left[1+\left(\frac{DIM}{B}*rate\right)\right]-\left[\frac{price}{100}+\left(\frac{A}{B}*rate\right)\right]}{\frac{price}{100}+\left(\frac{A}{B}*rate\right)}\right\}*\left(\frac{B}{DSM}\right)$$

In the equation above, *DIM* represents the number of days in the period starting with the issue date and ending with the maturity date. *DSM* represents the number of days in the period starting with the settlement date and ending with the maturity date. *A* represents the number of days in the period starting with the issue date and ending with the settlement date. *B* represents the number of days in a year based on the annual basis.

Example

In this example, YIELD_MATURITY computes the annual yield of a security that pays interest at maturity which is selling at $95.40663 with the settlement date of August 1, 2000, the issue date of July 1, 2000, the maturity date of July 1, 2010, and the interest rate of 6% at the issue using the US (NASD) 30/360 day count method.

```
settlement = VAR_TO_DT(2000, 8, 1)

maturity = VAR_TO_DT(2010, 7, 1)

issue = VAR_TO_DT(2000, 7, 1)

rate = .06

price = 95.40663

basis = 1

PRINT, YIELD_MATURITY(settlement, maturity, issue, $
                       rate, price, basis)

     0.0673905
```

YIELD_PERIODIC Function

Evaluates the yield of a security that pays periodic interest.

Usage

result = YIELD_PERIODIC (*settlement, maturity, coupon_rate, price, redemption, frequency, basis*)

Input Parameters

settlement — The date on which payment is made to settle a trade. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

maturity — The date on which the bond comes due, and principal and accrued interest are paid. For a more detailed discussion on dates see Chapter 8, *Working with Date/Time Data* in the PV-WAVE User's Guide.

coupon_rate — Annual coupon rate.

price — Price per $100 face value of the security.

redemption — Redemption value per $100 face value of the security.

frequency — Frequency of the interest payments. It should be either 1, 2 or 4.

frequency	Meaning
1	One payment per year (Annual payment)
2	Two payments per year (Semi-annual payment)
4	Four payments per year (Quarterly payment)

basis — The method for computing the number of days between two dates. It should be either 0, 1, 2, 3 or 4.

basis	Day count basis
0	Actual/Actual
1	US (NASD) 30/360
2	Actual/360
3	Actual/365
4	European 30/360

Returned Value

result **—** The yield of a security that pays interest periodically. If no result can be computed, NaN is returned.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Xguess **—** If present, the value is used as the initial guess at the internal rate of return.

Highest **—** If present, the value is used as the maximum value of the internal rate of return allowed.

Discussion

Function YIELD_PERIODIC computes the yield of a security that pays periodic interest. If there is one coupon period use the following:

$$\left\{\frac{\left(\frac{redemption}{100}+\frac{coupon\_rate}{frequency}\right)-\left[\frac{price}{100}+\left(\frac{A}{E}*\frac{coupon\_rate}{frequency}\right)\right]}{\frac{price}{100}+\left(\frac{A}{E}*\frac{coupon\_rate}{frequency}\right)}\right\}\left(\frac{frequency*E}{DSR}\right)$$

In the equation above, *DSR* represents the number of days in the period starting with the settlement date and ending with the redemption date. *E* represents the number of days within the coupon period. *A* represents the number of days in the period starting with the beginning of coupon period and ending with the settlement date.

If there is more than one coupon period use the following:

$$price - \left(\left(\frac{redemption}{\left(1+\frac{yield}{frequency}\right)^{\left(N-1+\frac{DSC}{E}\right)}}\right) + \left[\sum_{k=1}^{N} \frac{100*\frac{rate}{frequency}}{\left(1+\frac{yield}{frequency}\right)^{\left(k-1+\frac{DSC}{E}\right)}}\right] - \left(100*\frac{rate}{frequency}*\frac{A}{E}\right)\right) = 0$$

In the equation above, *DSC* represents the number of days in the period from the settlement to the next coupon date. *E* represents the number of days within the coupon period. *N* represents the number of coupons payable in the period starting with the settlement date and ending with the redemption date. *A* represents the number of days in the period starting with the beginning of the coupon period and ending with the settlement date.

Example

In this example, YIELD_PERIODIC computes yield of a security which is selling at $95.40663 with the settlement date of July 1, 1985, the maturity date of July 1, 1995, and the coupon rate of 6% at the issue using the US (NASD) 30/360 day count method.

```
settlement = VAR_TO_DT(2000, 7, 1)
maturity = VAR_TO_DT(2010, 7, 1)
coupon_rate = .06
price = 95.40663
redemption = 105.
frequency = 2
basis = 1
PRINT, YIELD_PERIODIC(settlement, maturity, coupon_rate, $
                       price, redemption, frequency, basis)
     0.0700047
```

Chapter 10: Basic Statistics and Random Number

FAURE_INIT Function

Initializes the structure used for computing a shuffled Faure sequence.

Usage

result = FAURE_INIT(*ndim*)

Input Parameters

ndim — The dimension of the hyper-rectangle.

Returned Value

A structure that contains information about the sequence.

Input Keywords

Base — The base of the Faure sequence.
Default: The smallest prime greater than or equal to *ndim*.

Skip — The number of points to be skipped at the beginning of the Faure sequence. Default:

$$\left\lfloor base^{m/2-1} \right\rfloor$$

where

$$m = \left\lfloor \log B / \log base \right\rfloor$$

and *B* is the largest representable integer.

Discussion

Discrepancy measures the deviation from uniformity of a point set.

The discrepancy of the point set

$$x_1, \ldots, x_n \in [0,1]^d \text{ , } d \geq 1,$$

is

$$D_n^{(d)} = \sup_E \left| \frac{A(E;n)}{n} - \lambda(E) \right|,$$

where the supremum is over all subsets of $[0, 1]^d$ of the form

$$E = [0, t_1) \times \ldots \times [0, t_d), \; 0 \leq t_j \leq 1, \; 1 \leq j \leq d,$$

λ is the Lebesque measure, and

$$(E;n$$

is the number of the x_j contained in E.

The sequence x_1, x_2, … of points $[0,1]^d$ is a low-discrepancy sequence if there exists a constant $c(d)$, depending only on d, such that

$$D_n^{(d)} \leq c(d) \frac{(\log n)^d}{n}$$

for all $n>1$.

Generalized Faure sequences can be defined for any prime base $b \geq d$. The lowest bound for the discrepancy is obtained for the smallest prime $b \geq d$, so the keyword *Base* defaults to the smallest prime greater than or equal to the dimension.

The generalized Faure sequence x_1, x_2, …, is computed as follows:

Write the positive integer n in its *b-ary* expansion,

$$n = \sum_{i=0}^{\infty} a_i(n) b^i$$

where $a_i\,(n)$ are integers,

$$0 \le a_i(n) < b$$

The *j-th* coordinate of x_n is

$$x_n^{(j)} = \sum_{k=0}^{\infty}\sum_{d=0}^{\infty} c_{kd}^{(j)}\; a_d\,(n)\; b^{-k-1}, \qquad 1 \le j \le d$$

The generator matrix for the series,

$$c_{k\,d}^{(j)}$$

is defined to be

$$c_{k\,d}^{(j)} = j^{\,d-k} c_{k\,d}$$

and

$$c_{k\,d}$$

is an element of the Pascal matrix,

$$c_{k\,d} = \begin{cases} \dfrac{d!}{c!(d-c)!} & k \le d \\ 0 & k > d \end{cases}$$

It is faster to compute a shuffled Faure sequence than to compute the Faure sequence itself. It can be shown that this shuffling preserves the low-discrepancy property.

The shuffling used is the *b-ary* Gray code. The function *G(n)* maps the positive integer *n* into the integer given by its *b-ary* expansion.

The sequence computed by this function is *x*(*G*(*n*)), where *x* is the generalized Faure sequence.

Example

In this example, five points in the Faure sequence are computed. The points are in the three-dimensional unit cube.

Note that FAURE_INIT is used to create a structure that holds the state of the sequence. Each call to FAURE_NEXT_PT returns the next point in the sequence and updates the state structure.

```
state = FAURE_INIT(3)

p = FAURE_NEXT_PT(5, state)

PM, p

      0.333689      0.492659     0.0640654

      0.667022      0.825992      0.397399

      0.778133      0.270436      0.175177

      0.111467      0.603770      0.508510

      0.444800      0.937103      0.841843
```

FAURE_NEXT_PT Function

Computes a shuffled Faure sequence.

Usage

result = FAURE_NEXT_PT(*npts, state*)

Input Parameters

npts **—** The number of points to generate in the hyper-rectangle.

state **—** State structure created by a call to FAURE_INIT.

Returned Value

An array of size *npts* by *state.dim* containing the *npts* next points in the shuffled Faure sequence.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Output Keywords

Skip **—** The current point in the sequence. The sequence can be restarted by initializing a new sequence using this value for *Skip*, and using the same dimension for *ndim*.

Discussion

Discrepancy measures the deviation from uniformity of a point set.

The discrepancy of the point set

$$x_1, \ldots, x_n \in [0,1]^d, d \geq 1,$$

is

$$D_n^{(d)} = \sup_E \left| \frac{A(E;n)}{n} - \lambda(E) \right|,$$

where the supremum is over all subsets of [0, 1]^d of the form

$$E = [0, t_1) \times \ldots \times [0, t_d),\ 0 \leq t_j \leq 1,\ 1 \leq j \leq d,$$

λ is the Lebesque measure, and

$$(E;n$$

is the number of the x_j contained in *E*.

The sequence x_1, x_2, … of points [0,1]^d is a low-discrepancy sequence if there exists *a* constant *c(d)*, depending only on *d,* such that

$$D_n^{(d)} \leq c(d) \frac{(\log n)^d}{n}$$

for all *n*>1.

Generalized Faure sequences can be defined for any prime base *b*≥*d*. The lowest bound for the discrepancy is obtained for the smallest prime *b*≥*d*, so the keyword Base defaults to the smallest prime greater than or equal to the dimension.

The generalized Faure sequence $x_1, x_2, \ldots$, is computed as follows:

Write the positive integer n in its *b-ary* expansion

$$n = \sum_{i=0}^{\infty} a_i(n) b^i$$

where $a_i(n)$ are integers,

$$0 \le a_i(n) < b$$

The j-th coordinate of x_n is

$$x_n^{(j)} = \sum_{k=0}^{\infty} \sum_{d=0}^{\infty} c_{kd}^{(j)} \; a_d(n) \; b^{-k-1}, \qquad 1 \le j \le d$$

The generator matrix for the series,

$$c_{k\,d}^{(j)},$$

is defined to be

$$c_{k\,d}^{(j)} = j^{\,d-k} c_{k\,d}$$

and

$$c_{k\,d}$$

is an element of the Pascal matrix,

$$c_{k\,d} = \begin{cases} \dfrac{d!}{c!(d-c)!} & k \le d \\ 0 & k > d \end{cases}$$

It is faster to compute a shuffled Faure sequence than to compute the Faure sequence itself. It can be shown that this shuffling preserves the low-discrepancy property.

The shuffling used is the *b-ary* Gray code. The function *G(n)* maps the positive integer *n* into the integer given by its *b-ary* expansion.

The sequence computed by this function is *x*(*G(n)*), where *x* is the generalized Faure sequence.

Example

In this example, five points in the Faure sequence are computed. The points are in the three-dimensional unit cube.

Note that FAURE_INIT is used to create a structure that holds the state of the sequence. Each call to FAURE_NEXT_PT returns the next point in the sequence and updates the state structure.

```
state = FAURE_INIT(3)
p = FAURE_NEXT_PT(5, state)
PM, p
      0.333689      0.492659     0.0640654
      0.667022      0.825992      0.397399
      0.778133      0.270436      0.175177
      0.111467      0.603770      0.508510
      0.444800      0.937103      0.841843
```

New PV-WAVE:IMSL Statistics Commands

This section lists the new functions and procedures have been added to PV-WAVE:IMSL Statistics for version 7.5.

Chapter 8: Time Series and Forecasting

KALMAN Procedure

Performs Kalman filtering and evaluates the likelihood function for the state-space model.

Usage

KALMAN, *b*, *covb*, *n*, *ss*, *alndet*

Input/Output Parameters

b — One dimensional array of containing the estimated state vector. The input is the estimated state vector at time *k* given the observations through time *k* – 1. The output is the estimated state vector at time *k* + 1 given the observations through time *k*. On the first call to KALMAN, the input *b* must be the prior mean of the state vector at time.

covb — Two dimensional array of size N_ELEMENTS(*b*) by N_ELEMENTS(*b*) such that $covb * \sigma^2$ is the mean squared error matrix for *b*. Before the first call to KALMAN, $covb * \sigma^2$ must equal the variance-covariance matrix of the state vector.

n — Named vaiable containing the rank of the variance-covariance matrix for all the observations. n must be initialized to zero before the first call to KALMAN. In the usual case when the variance-covariance matrix is nonsingular, *n* equals the sum of the N_ELEMENTS(Y) from the invocations to KALMAN. See the keyword section below for the definition of *Y*.

ss — Named vaiable containing the generalized sum of squares. *ss* must be initialized to zero before the first call to KALMAN. The estimate of σ^2 is given by

$$\frac{ss}{n}.$$

alnet — Named vaiable containing the natural log of the product of the nonzero eigenvalues of *P* where $P * \sigma^2$ is the variance-covariance matrix of the observations. Although *alndet* is computed, KALMAN avoids the explicit computation of *P*. alndet must be initialized to zero before the first call to KALMAN. In the usual case when *P* is nonsingular, *alndet* is the natural log of the determinant of *P*.

Input Keywords

Y — One dimensional array containing the observations. Keywords *Y*, *Z* and *R* indicate an update step and must be used together

R — Two dimensional array if size N_ELEMENTS(*Y*) by N_ELEMENTS(Y) containing the matrix such that $R * \sigma^2$ is the variance-covariance matrix of errors in the observation equation. Keywords *Y*, *Z* and *R* indicate an update step and must be used together.

T_matrix — Two dimensional array if size N_ELEMENTS(*b*) by N_ELEMENTS(b) containing the transition matrix in the state equation.

Default: *T_matrix = identity matrix*

Q_matrix — Two dimensional array if size N_ELEMENTS(*b*) by N_ELEMENTS(*b*) matrix such that $Q\_matrix * \sigma^2$ is the variance-covariance matrix of the error vector in the state equation.

Default: There is no error term in the state equation

Tolerance — Tolerance used in determining linear dependence.

Default: Tolerance = 100*eps where eps is machine precision.

Output Keywords

V — One dimensional array of length N_ELEMENTS(*Y*) containing the one-step-ahead prediction error.

Covv — Two dimensional array if size N_ELEMENTS(*Y*) by N_ELEMENTS(*Y*) containing a matrix such that $Covv * \sigma^2$ is the variance-covariance matrix of *v*.

Discussion

Routine KALMAN is based on a recursive algorithm given by Kalman (1960), which has come to be known as the Kalman filter. The underlying model is known as the state-space model. The model is specified stage by stage where the stages generally correspond to time points at which the observations become available. The routine KALMAN avoids many of the computations and storage requirements that would be necessary if one were to process all the data at the end of each stage in order to estimate the state vector. This is accomplished by using previous computations and retaining in storage only those items essential for processing of future observations.

The notation used here follows that of Sallas and Harville (1981). Let y_k (input in keyword *Y*) be the $n_k \times 1$ vector of observations that become available at time *k*. The subscript *k* is used here rather than *t*, which is more customary in time series, to emphasize that the model is expressed in stages $k = 1, 2, \ldots$ and that these stages need not correspond to equally spaced time points. In fact, they need not correspond to time points of any kind. The *observation equation* for the state-space model is

$$y_k = Z_k b_k + e_k \qquad k = 1, 2, \ldots$$

Here, Z_k is an $n_k \times q$ known matrix and b_k is the $q \times 1$ state vector. The state vector b_k is allowed to change with time in accordance with the *state equation*

$$b_{k+1} = T_{k+1} b_k + w_{k+1} \qquad k = 1, 2, \ldots$$

starting with $b_1 = \mu_1 + w_1$.

The change in the state vector from time k *to* $k + 1$ is explained in part by the *transition matrix* T_{k+1} (the identity matrix by default, or optionally input using keyword *T_MATRIX*), which is assumed known. It is assumed that the *q-dimensional* w_k*s* ($k = 1, 2, \ldots K$) are independently distributed multivariate normal with mean vector 0 and variance-covariance matrix $\sigma^2 Q_k$, that the n_k-dimensional e_k*s* ($k = 1, 2, \ldots K$) are independently distributed multivariate normal with mean vector 0 and variance-covariance matrix $\sigma^2 R_k$, and that the w_k*s* and e_k*s* are independent of each other. Here, $\mu_1$is the mean of b_1 and is assumed known, σ^2 is an unknown positive scalar. Q_{k+1} (input in *Q)* and R_k (input in keyword *R)* are assumed known.

Denote the estimator of the realization of the state vector b_k given the observations $y_1, y_2, \ldots, y_j$ by

$$\hat{\beta}_{k|j}$$

By definition, the mean squared error matrix for

$$\hat{\beta}_{k|j}$$

is

$$\sigma^2 C_{k|j} = E(\hat{\beta}_{k|j} - b_k)(\hat{\beta}_{k|j} - b_k)^T$$

At the time of the *k*-th invocation, we have

$$\hat{\beta}_{k|k-1}$$

and

$C_{k|k-1}$, which were computed from the (*k*–1)-st invocation, input in *b* and *covb*, respectively. During the *k*-th invocation, routine KALMAN computes the filtered estimate

$$\hat{\beta}_{k|k}$$

along with $C_{k|k}$. These quantities are given by the *update equations*:

$$\hat{\beta}_{k|k} = \hat{\beta}_{k|k-1} + C_{k|k-1} Z_k^T H_k^{-1} v_k$$
$$C_{k|k} = C_{k|k-1} - C_{k|k-1} Z_k^T H_k^{-1} Z_k C_{k|k-1}$$

where

$$v_k = y_k - Z_k \hat{\beta}_{k|k-1}$$

and where

$$H_k = R_k + Z_k C_{k|k-1} Z_k^T$$

Here, v_k (stored in *v*) is the one-step-ahead prediction error, and $\sigma^2 H_k$ is the variance-covariance matrix for v_k. H_k is stored in *covv*. The “start-up values” needed on the first invocation of KALMAN are

$$\hat{\beta}_{1|0} = \mu_1$$

and $C_{1|0} = Q_1$ input via *b* and *covb*, respectively. Computations for the *k*-th invocation are completed by KALMAN computing the one-step-ahead estimate

$$\hat{\beta}_{k+1|k}$$

along with $C_{k+1|k}$ given by the *prediction equations:*

$$\hat{\beta}_{k+1|k} = T_{k+1}\hat{\beta}_{k|k}$$
$$C_{k+1|k} = T_{k+1}C_{k|k}T_{k+1}^{T} + Q_{k+1}$$

If both the filtered estimates and one-step-ahead estimates are needed by the user at each time point, KALMAN can be invoked twice for each time point—first without *T_matrix* and *Q_matrix* to produce

$$\hat{\beta}_{k|k}$$

and $C_{k|k}$, and second without keywords *Y, Z, and R* to produce

$$\hat{\beta}_{k+1|k}$$

and $C_{k+1|k}$ (Without *T_matrix* and *Q_matrix,* the prediction equations are skipped. Without keywords *Y, Z,* and *R,* the update equations are skipped.).

Often, one desires the estimate of the state vector more than one-step-ahead, i.e., an estimate of

$$\hat{\beta}_{k|j}$$

is needed where $k > j + 1$. At time j, KALMAN is invoked with keywords *Y, Z,* and *R* to compute

$$\hat{\beta}_{j+1|j}$$

Subsequent invocations of KALMAN without keywords *Y, Z,* and *R* can compute

$$\hat{\beta}_{j+2|j}, \hat{\beta}_{j+3|j}, \ldots, \hat{\beta}_{k|j}$$

Computations for

$$\hat{\beta}_{k|j}$$

and $C_{k|j}$ assume the variance-covariance matrices of the errors in the observation equation and state equation are known up to an unknown positive scalar multiplier, σ^2. The maximum likelihood estimate of σ^2 based on the observations $y_1, y_2, \ldots, y_m$, is given by

$$\hat{\sigma}^2 = SS / N$$

where

$$N = \sum_{k=1}^{m} n_k \; and \; SS = \sum_{k=1}^{m} v_k^T H_k^{-1} v_k$$

N and *SS* are the input/output arguments *n* and *ss*.

If σ^2 is known, the $R_k s$ *and* $Q_k s$ can be input as the variance-covariance matrices exactly. The earlier discussion is then simplified by letting $\sigma^2 = 1$.

In practice, the matrices T_k, Q_k, *and* R_k *a*re generally not completely known. They may be known functions of an unknown parameter vector θ. In this case, KALMAN can be used in conjunction with an optimization program (see routine FMINV, PV-WAVE: IMSL Mathematics Reference, Chapter 8, “Optimization”) to obtain a maximum likelihood estimate of θ. The natural logarithm of the likelihood function for $y_1, y_2, \ldots, y_m$ differs by no more than an additive constant from

$$\begin{aligned} L(\theta,\sigma^2; y_1, y_2, \quad , y_m) \quad & \frac{1}{2} N \quad \sigma \\ & \frac{1}{2} \sum_{k \quad 1}^{m} \quad [\quad (H_k)] \quad \frac{1}{2} \sigma \;\, ^2 \sum_{k \quad 1}^{m} v_k^T H_k \, ^1 v_k \end{aligned}$$

(Harvey 1981, page 14, equation 2.21).

Here,

(stored in *alndet*) is the natural logarithm of the determinant of V where $\sigma^2 V$ is the variance-covariance matrix of the observations.

Minimization of $-2L(\theta, \sigma^2; y_1, y_2, \ldots, y_m)$ over all θ and σ^2 produces maximum likelihood estimates. Equivalently, minimization of $-2L_c(\theta; y_1, y_2, \ldots, y_m)$ where

$$L_c(\theta; y_1, y_2, \ldots, y_m) = -\frac{1}{2} N \, ln\left(\frac{SS}{N}\right) - \frac{1}{2}\sum_{k=1}^{m} ln[det(H_k)]$$

produces maximum likelihood estimates

$$\hat{\theta} \text{ and } \hat{\sigma}^2 = SS / N$$

The minimization of $-2L_c(\theta; y_1, y_2, \ldots, y_m)$ instead of $-2L(\theta, \sigma^2; y_1, y_2, \ldots, y_m)$, reduces the dimension of the minimization problem by one. The two optimization problems are equivalent since

$$\hat{\sigma}^2(\theta) = SS(\theta) / N$$

minimizes $-2L(\theta, \sigma^2; y_1, y_2, \ldots, y_m)$ *for all* θ, consequently,

$$\hat{\sigma}^2(\theta)$$

can be substituted for σ^2 in $L(\theta, \sigma^2; y_1, y_2, \ldots, y_m)$ to give a function that differs by no more than an additive constant from $L_c(\theta; y_1, y_2, \ldots, y_m)$.

The earlier discussion assumed H_k to be nonsingular. If H_k is singular, a modification for singular distributions described by Rao (1973, pages 527–528) is used. The necessary changes in the preceding discussion are as follows:

1. Replace

$$H_k^{-1}$$

by a generalized inverse.

2. Replace det(H_k) by the product of the nonzero eigenvalues of H_k.

3. Replace *N by*

$$\sum_{k=1}^{m} rank\left(H_k\right)$$

Maximum likelihood estimation of parameters in the Kalman filter is discussed by Sallas and Harville (1988) and Harvey (1981, pages 111–113).

Example 1

Routine KALMAN is used to compute the filtered estimates and one-step-ahead estimates for a scalar problem discussed by Harvey (1981, pages 116–117). The observation equation and state equation are given by

$$y_k = b_k + e_k$$
$$b_{k+1} = b_k + w_{k+1} \qquad k = 1,2,3,4$$

where the e_k*s* are identically and independently distributed normal with mean 0 and variance σ^2, the w_k*s* are identically and independently distributed normal with mea*n* 0 and variance $4\sigma^2$, and b_1 is distributed normal with mean 4 and variance $16\sigma^2$. Two invocations of KALMAN are needed for each time point in order to compute the filtered estimate and the one-step-ahead estimate. The first invocation does not use the keywords *T_matrix* and *Q*_matrix so that the prediction equations are skipped in the computations. The update equations are skipped in the computations in the second invocation.

This example also computes the one-step-ahead prediction errors. Harvey (1981, page 117) contains a misprint for the value v_4 that he gives as 1.197. The correct value of v_4 = 1.003 is computed by KALMAN.

Note that this example is in the form of a WAVE procedure, with the output following the procedure.

```
PRO EX_KALMAN

z = 1

r = 1

q = 4

t = 1
```

```
b = 4
covb = 16

ydata = [4.4, 4, 3.5, 4.6]

n = 0
ss = 0
alndet = 0
format = "(2I4, 2F8.3, I4, 4F8.3)"
PRINT, "   k   j     b      covb   n    ss     alndet    v
   covv"
FOR i = 0, 3 DO BEGIN
   y = ydata(i)
   ; Update
   kalman, b, covb, n, ss, alndet, $
     Y = y, Z = Z, R = r, $
     v = v, covv = covv
   PRINT, i, i, b, covb, n, ss, alndet, v, covv, format =
   format

   ; Predict
   kalman, b, covb, n, ss, alndet, $
     t_matrix = t, q = q
   PRINT, i+1, i, b, covb, n, ss, alndet, v, covv, format =
   format

END

END
```

Output

```
k    j       b        covb    n       ss     alndet       v      covv
0    0    4.376      0.941    1    0.009      2.833    0.400    17.000
1    0    4.376      4.941    1    0.009      2.833    0.400    17.000
1    1    4.063      0.832    2    0.033      4.615   -0.376     5.941
```

```
2    1    4.063      4.832    2    0.033     4.615   -0.376     5.941
2    2    3.597      0.829    3    0.088     6.378   -0.563     5.832
3    2    3.597      4.829    3    0.088     6.378   -0.563     5.832
3    3    4.428      0.828    4    0.260     8.141    1.003      5.829
4    3    4.428      4.828    4    0.260     8.141    1.003      5.829
```

Chapter 11: Probability Distribution Functions and Inverses

BINOMIALPDF Function

Evaluates the binomial probability function.

Usage

result = BINOMIALPDF (*k*, *n*, *p*)

Input Parameters

***k* —** Argument for which the binomial probability function is to be evaluated.

***n* —** Number of Bernoulli trials.

***p* —** Probability of success on each trial.

Returned Value

***result* —** The probability that a binomial random variable takes a value equal to *k*.

Discussion

The function BINOMIALPDF evaluates the probability that a binomial random variable with parameters *n and p* takes on the value *k*. It does this by computing probabilities of the random variable taking on the values in its range less than (or the values greater than) *k*. These probabilities are computed by the recursive relationship

$$\Pr(X = j) = \frac{(n+1-j)p}{j(1-p)} \Pr(X = j-1)$$

To avoid the possibility of underflow, the probabilities are computed forward from 0, if *k* is not greater than *n* times *p*, and are computed backward from *n*, otherwise. The smallest positive machine number, ε, is used as the starting value for computing the probabilities, which are rescaled by $(1 - p)^n\varepsilon$ if forward computation is performed and by $p^n\varepsilon$ if backward computation is done.

For the special case of *p* = 0, BINOMIALPDF returns 0 if *k* is greater than 0 and to 1 otherwise; and for the case *p* = 1, BINOMIALPDF returns 0 if *k* is less than *n* and to 1 otherwise.

Example

Suppose *X* is a binomial random variable with *n* = 5 and *p* = 0.95. In this example, we find the probability that *X* is equal to 3.

```
PRINT, BINOMIALPDF(3, 5, .95)
```

```
0.0214344
```

Chapter 12: Random Number Generation

RANDOM_TABLE Procedure

Sets or retrieves the current table used in either the shuffled or GFSR random number generator.

Usage

```
RANDOM_TABLE, table, /Get
```

```
RANDOM_TABLE, table, /Set
```

Input/Output Parameters

table **—** One dimensional array used in the generators. For the shuffled generators table is length 128. For the GFSR generator table is length 1565. The argument *table* is input if the keyword *Set* is used, and output if the keyword *Get* is used.

Input Keywords

Set **—** If present and nonzero, then the specified table is being set.

Get **—** If present and nonzero, then the specified table is being retieved.

Gfsr **—** If present and nonzero, then the specified GFSR table is being set or retrieved.

Double **—** If present and nonzero, double precision is used. This keyword is active only when the shuffled table is being set or retrieved.

Discussion

The values in *table* are initialized by the IMSL random number generators. The values are all positive except if the user wishes to reinitialize the array, in which case the first element of the array is input as a nonpositive value. (Usually, one should avoid reinitializing these arrays, but it might be necessary sometimes in restarting a simulation.) If the first element of *table* is set to a nonpositive value on the call to RANDOM_TABLE with the keyword *Set*, on the next invocation of a routine to generate random numbers, the appropriate table will be reinitialized.

For more details on the shuffled and GFSR generators see the *Introduction* to Chapter 12 in the *PV-WAVE: IMSL Statistics Reference*.

Example

In this example, three separate simulation streams are used, each with a different form of the generator. Each stream is stopped and restarted. (Although this example is obviously an artificial one, there may be reasons for maintaining separate streams and stopping and restarting them because of the nature of the usage of the random numbers coming from the separate streams.)

```
nr = 5

iseed1 = 123457

iseed2 = 123457
```

```
iseed7 = 123457

; Begin first stream, iopt = 1 (by default)
RANDOMOPT,  Set = iseed1
r = RANDOM(nr)
RANDOMOPT,  Get = iseed1
PM, r, Title = 'First stream output'
First stream output
     0.966220
     0.260711
     0.766262
     0.569337
     0.844829
PRINT, 'output seed ', iseed1
output seed   1814256879

; Begin second stream, iopt = 2
RANDOMOPT,  gen_opt = 2
RANDOMOPT,  Set = iseed2
r = RANDOM(nr)
RANDOMOPT,  Get = iseed2
RANDOM_TABLE, table, /Get
PM, r, Title = 'Second stream output'
Second stream output
     0.709518
     0.186145
     0.479442
     0.603839
     0.379015
PRINT, 'output seed ', iseed2
output seed   1965912801


; Begin third stream, iopt = 7
RANDOMOPT,  gen_opt = 7
RANDOMOPT,  Set = iseed7
```

```
r = RANDOM(nr)
RANDOMOPT,  Get = iseed7
RANDOM_TABLE, itable, /Get, /GFSR
PM, r, Title = 'Third stream output'
Third stream output
   0.391352
   0.0262676
   0.762180
   0.0280987
   0.899731
PRINT, 'output seed ', iseed7
output seed   1932158269

; Reinitialize seed and resume first stream
RANDOMOPT,  gen_opt = 1
RANDOMOPT,  Set = iseed1
r = RANDOM(nr)
RANDOMOPT,  Get = iseed1
pm, r, title = 'First stream output'
First stream output
   0.0442665
   0.987184
   0.601350
   0.896375
   0.380854
PRINT, 'output seed ', iseed1
output seed    817878095


; Reinitialize seed and table for shuffling and
; resume second stream
RANDOMOPT,  gen_opt = 2
RANDOMOPT,  Set = iseed2
RANDOM_TABLE, table, /Set
r = RANDOM(nr)
RANDOMOPT,  Get = iseed2
```

```
PM, r, Title = 'Second stream output'
Second stream output
   0.255690
   0.478770
   0.225802
   0.345467
   0.581051
PRINT, 'output seed ', iseed2
output seed   2108806573

; Reinitialize seed and table for GFSR and
; resume third stream.
RANDOMOPT,  gen_opt = 7
RANDOMOPT,  Set = iseed7
RANDOM_TABLE, itable, /Set, /Gfsr
r = RANDOM(nr)
RANDOMOPT,  Get = iseed7
PM, r, Title = 'Third stream output'
Third stream output
   0.751854
   0.508370
   0.906986
   0.0910035
   0.691663
PRINT, 'output seed ', iseed7
output seed   1485334679
```

RANDOM_NPP Function

Generates pseudorandom numbers from a nonhomogeneous Poisson process.

Usage

result = RANDOM_NPP(*tbegin, tend, ftheta, theta_min, theta_max, neub*)

Input Parameters

tbegin — Lower endpoint of the time interval of the process.
tbegin must be nonnegative. Usually, *tbegin* = 0.

tend — Upper endpoint of the time interval of the process.
tend must be greater than *tbegin*.

ftheta — Scalar string specifying a user-supplied function to provide the value of the rate of the process as a function of time. This function accepts one argument and must be defined over the interval from *tbegin* to *tend* and must be nonnegative in that interval.

theta_min — Minimum value of the rate function ftheta() in the interval (*tbegin, tend*).
If the actual minimum is unknown, set *theta_min* = 0.0.

theta_max — Maximum value of the rate function *ftheta* in the interval (*tbegin, tend*).
If the actual maximum is unknown, set *theta_max* to a known upper bound of the maximum. The efficiency of RANDOM_NPP is less the greater *theta_max* exceeds the true maximum.

neub — Upper bound on the number of events to be generated.
In order to be reasonably sure that the full process through time *tend* is generated, calculate *neub* as *neub* = *X* + 10.0 * SQRT(X), where *X* = *theta_max* * (*tend* - *tbegin*).

Returned Value

A one dimensional array containing the times to events. If then length of the result is less that *neub,* the time *tend* is reached before *neub* events are realized

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Routine RANDOM_NPP simulates a one-dimensional nonhomogeneous Poisson process with rate function *theta* in a fixed interval (*tend* - *tbegin*].

Let $\lambda(t)$ be the rate function and t_0 = *tbegin* and t_1 = *tend*. Routine RANDOM_NPP uses a method of thinning a nonhomogeneous Poisson process

$\{N^*(t), t \geq t_0\}$ with rate function $\lambda^*(t) \geq \lambda(t)$ *in* $(t_0, t_1]$, where the number of events, N^*, in the interval $(t_0, t_1]$ has a Poisson distribution with parameter

$$\mu_0 = \int_t^{t_1} \lambda(t)\, dt$$

The function

$$\Lambda(t) = \int_0^{t'} \lambda(t)\, dt$$

is called the *integrated rate function.*In RANDOM_NPP, $\lambda^*(t)$ is taken to be a constant λ^*(= *theta_max*) so that at time t_i, the time of the next event t_{i+1} is obtained by generating and cumulating exponential random numbers

$$E^*_{1,i}, E^*_{2,i}, \ldots,$$

with parameter λ^*, until for the first time

$$u_{j,i} \leq \left(t_i + E^*_{1,i} + \cdots + E^*_{j,i}\right) / \lambda^*$$

where the $u_{j,i}$ are independent uniform random numbers between 0 and 1. This process is continued until the specified number of events, *neub*, is realized or until the time, *tend*, is exceeded. This method is due to Lewis and Shedler (1979), who also review other methods. The most straightforward (and most efficient) method is by inverting the integrated rate function, but often this is not possible.

If *theta_max* is actually greater than the maximum of $\lambda(t)$ in $(t_0, t_1]$, the routine will work, but less efficiently. Also, if $\lambda(t)$ varies greatly within the interval, the efficiency is reduced. In that case, it may be desirable to divide the time interval into subintervals within which the rate function is less variable. This is possible because the process is without memory.

If no time horizon arises naturally, *tend* must be set large enough to allow for the required number of events to be realized. Care must be taken, however, that *ftheta* is defined over the entire interval.

After simulating a given number of events, the next event can be generated by setting *tbegin* to the time of the last event (the sum of the elements in the result) and calling RANDOM_NPP again. Cox and Lewis (1966) discuss modeling applications of nonhomogeneous Poisson processes.

Example

In this example, RANDOM_NPP is used to generate the first five events in the time 0 to 20 (if that many events are realized) in a nonhomogeneous process with rate function

$$\lambda(t) = 0.6342\ e0.001427^{t}$$

for $0 < t \leq 20$.

Since this is a monotonically increasing function of t, the minimum is at $t = 0$ and is 0.6342, and the maximum is at $t = 20$ and is 0.6342 e0.02854 = 0.652561.

```
.RUN
- FUNCTION ftheta_npp, t
-    return, .6342*exp(.001427*t)
- END
% Compiled module: FTHETA_NPP.

randomopt, set=123457
neub = 5
tmax = .652561
tmin = .6342
tbegin=0
tend=20

r = RANDOM_NPP(tbegin, tend, 'ftheta_npp', tmin, tmax, neub)
PM, r
     0.0526598
     0.407979
     0.258399
     0.0197666
     0.167641
```

RANDOM_ORDER Function

Generates pseudorandom order statistics from a uniform (0, 1) distribution, or optionally from a standard normal distribution.

Usage

result = RANDOM_ORDER(*ifirst*, *ilast*, *n*)

Input Parameters

ifirst — First order statistic to generate.

ilast — Last order statistic to generate.
ilast must be greater than or equal to *ifirst*. The full set of order statistics from *ifirst* to *ilast* is generated. If only one order statistic is desired, set *ilast* = *ifirst*.

n — Size of the sample from which the order statistics arise.

Input Keywords

Double — If present and nonzero, double precision is used.

Uniform — If present and nonzero, generate pseudorandom order statistics from a uniform (0, 1) distribution. (Default)

Normal — If present and nonzero, generate pseudorandom order statistics from a standard normal distribution.

Returned Value

An array of length *ilast* + 1 – *ifirst* containing the random order statistics in ascending order.

The first element is the *ifirst* order statistic in a random sample of size *n* from the uniform (0, 1) distribution.

Discussion

Routine RANDOM_ORDER generates the *ifirst* through the *ilast* order statistics from a pseudorandom sample of size *n* from a uniform
(0, 1) distribution. Depending on the values of *ifirst* and *ilast*, different methods of generation are used to achieve greater efficiency. If *ifirst* = 1 and

ilast = *n*, that is, if the full set of order statistics are desired, the spacings between successive order statistics are generated as ratios of exponential variates. If the full set is not desired, a beta variate is generated for one of the order statistics, and the others are generated as extreme order statistics from conditional uniform distributions. Extreme order statistics from a uniform distribution can be obtained by raising a uniform deviate to an appropriate power.

Each call to RANDOM_ORDER yields an independent event. This means, for example, that if on one call the fourth order statistic is requested and on a second call the third order statistic is requested, the "fourth" may be smaller than the "third". If both the third and fourth order statistics from a given sample are desired, they should be obtained from a single call to RANDOM_ORDER (by specifying *ifirst* less than or equal to 3 and *ilast* greater than or equal to 4).

If the keyword *Normal* is present and nonzero, then RANDOM_ORDER generates the *ifirst* through the *ilast* order statistics from a pseudorandom sample of size *n*, from a normal (0, 1) distribution

Example

In this example, RANDOM_ORDER is used to generate the fifteenth through the nineteenth order statistics from a sample of size twenty.

```
r  =  random_order(15, 19, 20)
pm, r
       0.706909
       0.808627
       0.874552
       0.922146
       0.957402
```

RAND_TABLE_2WAY Function

Generates a pseudorandom two-way table.

Usage

result = RAND_TABLE_2WAY (*row_totals*, *col_totals*)

Input Parameters

row_totals — One dimensional array containing the row totals.

col_totals — One dimensional array containing the column totals. (Input)
The elements of *row_totals* and *col_totals* must be nonnegative and must sum to the same quantity.

Returned Value

A N_ELEMENTS(*row_totals*) by N_ELEMENTS(*col_totals*) random matrix with the given row and column totals.

Discussion

Routine RAND_TABLE_2WAY generates pseudorandom entries for a two-way contingency table with fixed row and column totals. The method depends on the size of the table and the total number of entries in the table. If the total number of entries is less than twice the product of the number of rows and columns, the method described by Boyette (1979) and by Agresti, Wackerly, and Boyette (1979) is used. In this method, a work vector is filled with row indices so that the number of times each index appears equals the given row total. This vector is then randomly permuted and used to increment the entries in each row so that the given row total is attained.

For tables with larger numbers of entries, the method of Patefield (1981) is used. This method can be considerably faster in these cases. The method depends on the conditional probability distribution of individual elements, given the entries in the previous rows. The probabilities for the individual elements are computed starting from their conditional means.

Example

In this example, RAND_TABLE_2WAY is used to generate a two by three table with row totals 3 and 5, and column totals 2, 4, and 2.

```
r  =  RAND_TABLE_2WAY([3, 5], [2, 4, 2])
PM, r
            2           1           0
            0           3           2
```

RAND_ORTH_MAT Function

Generates a pseudorandom orthogonal matrix or a correlation matrix.

Usage

result = RAND_ORTH_MAT(*n*)

Input Parameters

n — The order of the matrix to be generated.

Returned Value

A two-dimensional array containing the *n* by *n* random correlation matrix.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Eigenvalues **—** A one-dimensional array of length n containing the eigenvalues of the correlation matrix to be generated.   The elements of *Eigenvalues* must be positive, they must sum to *n*, and they cannot all be equal.

A_Matrix **—** A two-dimensional array containing *n* by *n* random orthogonal matrix. A random correlation matrix is generated using the orthogonal matrix input in *A_Matrix*. The keyword *Eigenvalues* must also be supplied if *A_Matrix* is used.

Discussion

Routine RAND_ORTH_MAT generates a pseudorandom orthogonal matrix from the invariant Haar measure. For each column, a random vector from a uniform distribution on a hypersphere is selected and then is projected onto the orthogonal complement of the columns already formed. The method is described by Heiberger (1978). (See also Tanner and Thisted 1982.)

If the keyword *Eigenvalues* is used, a correlation matrix is formed by applying a sequence of planar rotations to the matrix $A^T D^A$, *where D = diag(Eigenvalues(0),* …, *Eigenvalues*(*n*-*1*)), so as to yield ones along the diagonal. The planar rotations are applied in such an order that in the two by two matrix that determines the rotation, one diagonal element is less than 1.0 and one is greater than 1.0. This method is discussed by Bendel and Mickey (1978) and by Lin and Bendel (1985).

The distribution of the correlation matrices produced by this method is not known. Bendel and Mickey (1978) and Johnson and Welch (1980) discuss the distribution.

For larger matrices, rounding can become severe; and the double precision results may differ significantly from single precision results.

Example

In this example, RAND_ORTH_MAT is used to generate a 4 by 4 pseudorandom correlation matrix with eigenvalues in the ratio 1:2:3:4.

```
RANDOMOPT, set = 123457
a = RAND_ORTH_MAT(4)
ev = .4d0*[1.0d0, 2.0d0, 3.0d0, 4.0d0]
cor = RAND_ORTH_MAT(n, Eigenvalues = ev, A_Matrix= a)
PM, cor
      1.00000    -0.235786    -0.325795    -0.110139
    -0.235786      1.00000     0.190564   -0.0172391
    -0.325795     0.190564      1.00000    -0.435339
    -0.110139   -0.0172391    -0.435339      1.00000
```

RANDOM_SAMPLE Function

Generates a simple pseudorandom sample from a finite population.

Usage

result = RANDOM_SAMPLE(*nsamp*, *population*)

Input Parameters

nsamp — The sample size desired.

population — A one or two dimensional array containing the population to be sampled. If either of the keywords *First_Call* or *Additional_Call* are specified, then *population* contains a different part of the population on each invocation, otherwise *population* contains the entire population.

Returned Value

nsamp by *nvar* array containing the sample, where *nvar* is the number of columns in the argument *population*.

Input Keywords

Double — If present and nonzero, double precision is used.

First_Call — If present and nonzero, then this is the first invocation with this data; additional calls to RANDOM_SAMPLE may be made to add to the population. Additional calls should be made using the keyword *Additional_Call*. Keywords *Index* and *Npop* are required if *First_Call* is set. See Example 2 .

Additional_Call — If present and nonzero, then this is an additional invocation of RANDOM_SAMPLE, and updating for the subpopulation in *population* is performed. Keywords *Index, Npop* and *Sample* are required if *Additional_Call* is set. It is not necessary to know the number of items in the population in advance. *Npop* is used to cumulate the population size and should not be changed between calls to RANDOM_SAMPLE. See Example 2.

Input/Output Keywords

Index — A one-dimensional array of length *nsamp* containing the indices of the sample in the population. Output if keyword *First_Call* is used. Input/Output if keyword *Additional_Call* is used.

Npop — The number of items in the population. Output if keyword *First_Call* is used. Input/Output if keyword *Additional_Call* is used.

Sample — An array of size *nsamp* by *nvar* containing the sample. Initially, the result of calling RANDOM_SAMPLE with keyword *First_Call* is used for *Sample*.

Discussion

Routine RANDOM_SAMPLE generates a pseudorandom sample from a given population, without replacement, using an algorithm due to McLeod and Bellhouse (1983).

The first *nsamp* items in the population are included in the sample. Then, for each successive item from the population, a random item in the sample is replaced by that item from the population with probability equal to the sample size divided by the number of population items that have been encountered at that time.

Example 1

In this example, RANDOM_SAMPLE is used to generate a sample of size 5 from a population stored in the matrix *population*.

```
RANDOMOPT, Set = 123457

pop = STATDATA(2)

samp = RANDOM_SAMPLE(5, pop)

PM, samp

        1764.00       36.4000

        1828.00       62.5000

        1923.00       5.80000

        1773.00       34.8000

        1769.00       106.100
```

Example 2

Routine RANDOM_SAMPLE is now used to generate a sample of size 5 from the same population as in the example above except the data are input to RANDOM_SAMPLE one observation at a time. This is the way RANDOM_SAMPLE may be used to sample from a file on disk or tape. Notice that the number of records need not be known in advance.

```
RANDOMOPT, Set = 123457

pop = STATDATA(2)

samp = RANDOM_SAMPLE(5, pop(0, *), /First_Call, Index = ii,
   Npop=np)

FOR i=1,175 DO samp = RANDOM_SAMPLE(5, pop(i, *), /
   Additional_Call, $

        index = ii, npop = np, sample =  samp)

PM, samp

        1764.00       36.4000

        1828.00       62.5000

        1923.00       5.80000

        1773.00       34.8000

        1769.00       106.100
```

RAND_FROM_DATA Function

Generates pseudorandom numbers from a multivariate distribution determined from a given sample.

Usage

result = RAND_FROM_DATA(*n_random*, *x*, *nn*)

Input Parameters

n_random — Number of random multivariate vectors to generate.

x — Two dimensional array of size *nsamp* by *ndim* containing the given sample.

nn — Number of nearest neighbors of the randomly selected point in *x* that are used to form the output point in the result.

Returned Value

n by *ndim* matrix containing the random multivariate vectors in its rows.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Given a sample of size *nsamp* of observations of a *k-variate* random variable, RAND_FROM_DATA generates a pseudorandom sample with approximately the same moments as the given sample. The sample obtained is essentially the same as if sampling from a Gaussian kernel estimate of the sample density. (See Thompson 1989.) Routine RAND_FROM_DATA uses methods described by Taylor and Thompson (1986).

Assume that the (vector-valued) observations x_i are in the rows of *x*. An observation, x_j, is chosen randomly; its nearest *m* (= *nn*) *neighbors,*

$$x_{j_1}, x_{j_2}, \ldots, x_{j_m}$$

are determined; and the mean

$$\bar{x}_j$$

of those nearest neighbors is calculated. Next, a random sample

$u_1, u_2, \ldots, u_m$ is generated from a uniform distribution with lower bound

$$\frac{1}{m} - \sqrt{\frac{3(m-1)}{m^2}}$$

and upper bound

$$\frac{1}{m} + \sqrt{\frac{3(m-1)}{m^2}}$$

$\sum_{l=1}^{m} u_l\left(x_{jl} - \bar{x}_j\right) + \bar{x}_j$ delivered is

The process is then repeated until *n* such simulated variates are generated and stored in the rows of the result.

Example

In this example, RAND_FROM_DATA is used to generate 5 pseudorandom vectors of length 4 using the initial and final systolic pressure and the initial and final diastolic pressure from Data Set A in Afifi and Azen (1979) as the fixed sample from the population to be modeled. (Values of these four variables are in the seventh, tenth, twenty-first, and twenty-fourth columns of data set number nine in routine STATDATA, see Chapter 13: *Utilities* of this manual).

```
RANDOMOPT, Set = 123457
r = STATDATA(9)
x = FLTARR(113, 4)
x(*, 0) = r(*,6)
x(*, 1) = r(*,9)
x(*, 2) = r(*,20)
x(*, 3) = r(*,23)
r  =  RAND_FROM_DATA(5, x, 5)
PM, r
```

```
162.767      90.5057      153.717      104.877
153.353      78.3180      176.664      85.2155
93.6958      48.1675      153.549      71.3688
101.751      54.1855      113.121      56.2916
91.7403      58.7684      48.4368      28.0994
```

CONT_TABLE Procedure

Sets up table to generate pseudorandom numbers from a general continuous distribution.

Usage

CONT_TABLE, *f, iopt, ndata, table*

Input Parameters

***f* —** A scalar string specifying a user-supplied function to compute the cumulative distribution function. The argument to the function is the point at which the distribution function is to be evaluated.

***iopt* —** Indicator of the extent to which table is initialized prior to calling CONT_TABLE.

iopt	**Action**
0	CONT_TABLE fills the last four columns of table. The user inputs the points at which the CDF is to be evaluated in the first column of table. These must be in ascending order.
1	CONT_TABLE fills the last three columns of table. The user supplied function *f* is not used and may be a dummy function; instead, the cumulative distribution function is specified in the first two columns of table. The abscissas (in the first column) must be in ascending order and the function must be strictly monotonically increasing.

ndata **—** Number of points at which the CDF is evaluated for interpolation. *ndata* must be greater than or equal to 4.

Input/Output Parameters

table **—** *ndata* by 5 table to be used for interpolation of the cumulative distribution function.
The first column of *table* contains abscissas of the cumulative distribution function in ascending order, the second column contains the values of the CDF (which must be strictly increasing), and the remaining columns contain values used in interpolation. The first row of *table* corresponds to the left limit of the support of the distribution and the last row corresponds to the right limit of the support; that is, *table* (0, 1) = 0.0 and *table*(*ndata*-1, 1) = 1.0.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Discussion

Routine CONT_TABLE sets up a table that routine RAND_GEN_CONT (page 219) can use to generate pseudorandom deviates from a continuous distribution. The distribution is specified by its cumulative distribution function, which can be supplied either in tabular form in *table* or by a function *f*. See the documentation for the routine RAND_GEN_CONT for a description of the method.

Example

For an example of using CONT_TABLE see the example for routine RAND_GEN_CONT (page 219).

RAND_GEN_CONT Function

Generates pseudorandom numbers from a general continuous distribution.

Usage

result = RAND_GEN_CONT(*n*, *table*)

Input Parameters

n — Number of random numbers to generate.

table— A two-dimensional array setup using CONT_TABLE to be used for interpolation of the cumulative distribution function.
The first column of *table* contains abscissas of the cumulative distribution function in ascending order, the second column contains the values of the CDF (which must be strictly increasing beginning with 0.0 and ending at 1.0) and the remaining columns contain values used in interpolation.

Returned Value

An array of length *n* containing the random deviates.

Input Keywords

Double — If present and nonzero, double precision is used.

Discussion

Routine RAND_GEN_CONT generates pseudorandom numbers from a continuous distribution using the inverse CDF technique, by interpolation of points of the distribution function given in table, which is set up by routine CONT_TABLE (page 218). A strictly monotone increasing distribution function is assumed. The interpolation is by an algorithm attributable to Akima (1970), using piecewise cubics. The use of this technique for generation of random numbers is due to Guerra, Tapia, and Thompson (1976), who give a description of the algorithm and accuracy comparisons between this method and linear interpolation. The relative errors using the Akima interpolation are generally considered very good.

Example

In this example, RAND_GEN_CONT (page 219) is used to set up a table for generation of beta pseudorandom deviates. The CDF for this distribution is computed by the routine BETACDF (Chapter 11). The table contains 100 points at which the CDF is evaluated and that are used for interpolation. Notice that two warnings are issued during the computations for this example.

```
FUNCTION cdf, x
   return, BETACDF(x, 3., 2.)
```

```
END

iopt = 0
ndata = 100;
table = FLTARR(100, 5)
x = 0.0;
table(*,0) = FINDGEN(100)/100.
CONT_TABLE, ’cdf’, iopt, ndata, table
RANDOMOPT, Set = 123457

r = RAND_GEN_CONT(5, table)
% BETACDF: Note: STAT_ZERO_AT_X
   Since "X" = 0.000000e+00 is less than or equal to zero,
the distribution function is zero at "x".
% CONT_TABLE: Warning: STAT_SECOND_COL_TABLE3
CDF in the second column of table did not begin at 0.0
and end at 1.0, but they have been adjusted. Prior
to adjustment, table(0, 1) = 0.000000e+00 and
table(ndata-1, 1)= 9.994079e-01.

PM, r
       0.92079391
       0.46412855
       0.76678398
       0.65357975
       0.81706959
```

DISCR_TABLE Function

Sets up table to generate pseudorandom numbers from a general discrete distribution.

Usage

result = DISCR_TABLE(*prf*, *del*, *nndx*, *imin*, *nmass*)

Input Parameters

prf — A scalar string specifying a user-supplied function to compute the probability associated with each mass point of the distribution The argument to the function is the point at which the probability function is to be evaluated. The argument to the function can range from *imin* to the value at which the cumulative probability is greater than or equal to 1.0 – *del*.

del — Maximum absolute error allowed in computing the cumulative probabi ity.
Probabilities smaller than *del* are ignored; hence, *del* should be a small positive number. If *del* is too small, however, *cumpr* (*nmass*-1) must be exactly 1.0 since that value is compared to 1.0 – *del*.

nndx — The number of elements of *cumpr* available to be used as indexes. *nndx* must be greater than or equal to 1. In general, the larger *nndx* is, to within sixty or seventy percent of *nmass*, the more efficient the generation of random numbers using RAND_GEN_DISCR will be.

Input/Out Parameters

imin — Scalar containing the smallest value the random deviate can assume. By default, *prf* is evaluated at *imin*. If this value is less than *del*, *imin* is incremented by 1 and again *prf* is evaluated at *imin*. This process is continued until *prf*(*imin*) ≥ *del*. *imin* is output as this value and *result*(0) is output as *prf*(*imin*).

nmass — Scalar containing the number of mass points in the distribution.
Input, if keyword *CUM_probs* is used; otherwise, output.
By default, *nmass* is the smallest integer such that
$prf(imin + nmass - 1) > 1.0 - del$. *nmass* does include the points $imin_{in} + j$ for *which* $prf(imin_{in} + j) < del$, *for* $j = 0, 1, \ldots,$
$imin_{out} - imin_{in}$, where $imin_{in}$ denotes the input value of *imin* and $imin_{out}$ denotes its output value.

Returned Value

Array, *cumpr*, of length *nmass* + *nndx* containing in the first *nmass* positions, the cumulative probabilities and in some of the remaining positions, indexes to speed access to the probabilities.

Input Keywords

Double — If present and nonzero, double precision is used.

Cum_Probs — One dimensional array of length *nmass* containing the cumulative probabilities to be used in computing the index portion of the result. If the keyword *Cum_Probs* is used, *prf* is not used and may be a dummy function.

Discussion

Routine DISCR_TABLE sets up a table that routine RAND_GEN_CONT (page 219) uses to generate pseudorandom deviates from a discrete distribution. The distribution can be specified either by its probability function *prf* or by a vector of values of the cumulative probability function. Note that *prf* is not the cumulative probability distribution function. If the cumulative probabilities are already available in *Cum_Probs,* the only reason to call DISCR_TABLE *i*s to form an index vector in the upper portion of the result so as to speed up the generation of random deviates by the routine RAND_GEN_CONT.

Example 1

In this example, DISCR_TABLE is used to set up a table to generate pseudorandom variates from the discrete distribution:

$$Pr(X = 1) = .05$$

$$Pr(X = 2) = .45$$

$$Pr(X = 3) = .31$$

$$Pr(X = 4) = .04$$

$$Pr(X = 5) = .15$$

In this simple example, we input the cumulative probabilities directly using keyword *Cum_Probs* and request 3 indexes to be computed (*nndx* = 4). Since the number of mass points is so small, the indexes would not have much effect on the speed of the generation of the random variates.

```
function PRF, x
  return, 0
end
cum_probs = [.05, .5, .81, .85, 1]
cumpr = DISCR_TABLE('PRF', 0.00001, 4, 1, 5, cum_probs =
   cum_probs)
PM, cumpr
      0.0500000
      0.500000
      0.810000
      0.850000
       1.00000
       3.00000
       1.00000
       2.00000
       5.00000
```

Example 2

This example, DISCR_TABLE is used to set up a table to generate binomial variates with parameters 20 and 0.5. The routine BINOMIALPDF (Chapter 11, Probability Distribution and Inverses) is used to compute the probabilities.

```
FUNCTION PRF, ix
  RETURN,  BINOMIALPDF(ix, 20, .5)
END

cumpr = DISCR_TABLE('PRF', 0.00001, 12, 0, 21)
PM, cumpr
  1.90735e-05
  0.000200272
   0.00128746
   0.00590802
    0.0206938
    0.0576583
     0.131587
     0.251722
     0.411901
     0.588099
     0.748278
     0.868413
     0.942342
     0.979306
     0.994092
     0.998713
     0.999800
     0.999981
      1.00000
      11.0000
      1.00000
      7.00000
      8.00000
      9.00000
      9.00000
      10.0000
      11.0000
      11.0000
      12.0000
      13.0000
      19.0000
```

RAND_GEN_DISCR Function

Generates pseudorandom numbers from a general discrete distribution using an alias method or optionally a table lookup method.

Usage

result = RAND_GEN_DISCR(*n*, *imin*, *nmass*, *probs*)

Input Parameters

n — Number of random numbers to generate.

imin — Smallest value the random deviate can assume.
This is the value corresponding to the probability in probs(0).

nmass — Number of mass points in the discrete distribution.

probs — Array of length nmass containing probabilities associated with the individual mass points. The elements of probs must be nonnegative and must sum to 1.0.

If the keyword *Table* is used, then *probs* is a vector of length at least *nmass* + 1 containing in the first *nmass* positions the cumulative probabilities and, possibly, indexes to speed access to the probabilities.
Routine DISCR_TABLE (page 221) can be used to initialize probs properly. If no elements of probs are used as indexes, *probs* (*nmass*) is 0.0 on input. The value in *probs*(0) is the probability of *imin*. The value in *probs* (*nmass*-1) must be exactly 1.0 (since this is the CDF at the upper range of the distribution.)

Returned Value

An integer array of length *n* containing the random discrete deviates.

Input Keywords

Double — If present and nonzero, double precision is used.

Table — If present and nonzero, generate pseudorandom numbers from a general discrete distribution using a table lookup method. If this keyword is used, then *probs* is a vector of length at least *nmass* + 1 containing in the first *nmass* positions the cumulative probabilities and, possibly, indexes to speed access to the probabilities. Routine DISCR_TABLE (page 221) can be used to initialize *probs* properly.

Discussion

Routine RAND_GEN_DISCR generates pseudorandom numbers from a discrete distribution with probability function given in the vector *probs*; that is

$$Pr(X = i) = p_j$$

for $i = i_{min}, i_{min} + 1, \ldots, i_{min} + n_m - 1$ where $j = i - i_{min} + 1$, $p_j = probs(j)$, $i_{min} = imin$, and $n_m = nmass$.

The algorithm is the *alias* method, due to Walker (1974), with modifications suggested by Kronmal and Peterson (1979).

If the keyword *Table* is used, RAND_GEN_DISCR generates pseudorandom deviates from a discrete distribution, using the table *probs*, which contains the cumulative probabilities of the distribution and, possibly, indexes to speed the search of the table. The DISCR_TABLE (page 221) can be used to set up the table *probs*. RAND_GEN_DISCR uses the inverse CDF method to generate the variates.

Example 1

In this example, RAND_GEN_DISCR is used to generate five pseudorandom variates from the discrete distribution:

$$Pr(X = 1) = .05$$

$$Pr(X = 2) = .45$$

$$Pr(X = 3) = .31$$

$$Pr(X = 4) = .04$$

$$Pr(X = 5) = .15$$

```
probs = [.05, .45, .31, .04, .15]
n = 5
imin = 1
nmass = 5
RANDOMOPT, Set_seed = 123457
r = RAND_GEN_DISCR(n, imin, nmass, probs)
PM, r
             3
             2
             2
             3
             5
```

Example 2

In this example, DISCR_TABLE (page 221) is used to set up a table and then RAND_GEN_DISCR is used to generate five pseudorandom variates from the binomial distribution with parameters 20 and 0.5.

```
FUNCTION PRF, ix
  RETURN,  BINOMIALPDF(ix, 20, .5)
END
imin = 0
nmass = 21
RANDOMOPT, Set_seed = 123457
cumpr = DISCR_TABLE('prf', 0.00001, 12, imin, nmass)
```

```
r = RAND_GEN_DISCR(n, imin, nmass, cumpr, /table)
PM, r
          14
           9
          12
          10
          12
```

RANDOM_ARMA Function

Generates a time series from a specific ARMA model.

Usage

result = RANDOM_ARMA(*n*, *nparams*)

result = RANDOM_ARMA(*n*, *nparams*, *ar*)

result = RANDOM_ARMA(*n*, *nparams*, *ma*)

result = RANDOM_ARMA(*n*, *nparams*, *ar*, *ma*)

Input Parameters

n — Number of observations to be generated. Parameter *n* must be greater than or equal to one.

nparams — One-dimensional array containing the parameters *p* and *q* consecutively. *nparams*(0) = *p*, where *p* is the number of autoregressive parameters. Parameter *p* must be greater than or equal to zero. *nparams*(1) = *q*, where *q* is the number of moving average parameters. Parameter *q* must be greater than or equal to zero.

ar — One-dimensional array of length *p* containing the autoregressive parameters.

ma — One-dimensional array of length *q* containing the moving average parameters.

Returned Value

result — One-dimensional array of length *n* containing the generated time series.

Input Keywords

Double — If present and nonzero, double precision is used.

Const — Overall constant. See the *Discussion* section.

Default: *Const* = 0

Var_Noise — If present (and *Input_Noise* is *not* used), the noise a_t will be generated from a normal distribution with mean 0 and variance *Var_Noise*. Keywords *Var_Noise* and *Input_Noise* can not be used together.

Default: *Var_Noise* = 1.0

Input_Noise — One-dimensional array of length *n* + max (*Ar_Lags*(*i*)) containing the random noises. Keywords *Input_Noise* and *Var_Noise* can not be used together. Keywords *Input_Noise* and *Output_Noise* can not be used together.

Ar_Lags — One-dimensional array of length *p* containing the order of the nonzero autoregressive parameters.

Default: *Ar_Lags* = [1, 2, ..., *p*]

Ma_Lags — One-dimensional array of length *q* containing the order of the nonzero moving average parameters.

Default: *Ma_Lags* = [1, 2, ..., *q*]

W_Init — One-dimensional array of length max (*Ar_Lags*(*i*)) containing the initial values of the time series.

Default: *W_Init*(*) = *Const*/(1 − *ar*(0) − *ar*(1) − …− *ar*(*p* − 1))

Accept_Reject — If present and nonzero, the random noises will be generated from a normal distribution using an acceptance/rejection method. If keyword *Accept_Reject* is not used, the random noises will be generated using an inverse normal CDF method. This argument will be ignored if keyword *Input_Noise* is used.

Output Keywords

Output_Noise — Named variable into which a one-dimensional array of length *n* + max (*Ma_Lags*(*i*)) containing the random noises is stored.

Discussion

Function RANDOM_ARMA simulates an ARMA(*p*, *q*) process, $\{W_t\}$, for

$t = 1, 2, ..., n$. The model is

$$\phi(B)W_t = \theta_0 + \theta(B)A_t \qquad t \in Z$$

$$\phi(B) = 1 - \phi_1 B - \phi_2 B^2 - \ldots - \phi_p B^p$$
$$\theta(B) = 1 - \theta_1 B - \theta_2 B^2 - \ldots - \theta_q B^q$$

Let μ be the mean of the time series $\{W_t\}$. The overall constant θ_0 (*Const*) is

$$\theta_0 = \begin{cases} \mu & p = 0 \\ \mu\left(1 - \sum_{i=1}^{p} \phi_i\right) & p > 0 \end{cases}$$

Time series whose innovations have a nonnormal distribution may be simulated by providing the appropriate innovations in *Input_Noise* and start values in *W_Init*.

The time series is generated according to the following model:

X(*i*) = *Const* + *ar*(0) * X(i – *Ar_Lags*(0)) + … +

ar(*p* – 1) * *X*(*i* – *Ar_Lags*(*p* – 1)) +

A(*I*) – *ma*(0) * *A*(*i* – *Ma_Lags*(0)) – …–

ma(*q* – 1) * *A*(*i* – *Ma_Lags*(*q* – 1))

where the constant is related to the mean of the series,

$$\overline{W}$$

as follows:

$$Const = \overline{W} \cdot (1 - ar(0) - \ldots - ar(\mathrm{q} - 1))$$

and where

$X(t) = W(t),$ $\qquad t = 0, 1, \ldots, \mathrm{n} - 1$

and

$W(t) = \mathrm{W\_Init}(t + p),$ $\qquad t = -\mathrm{p}, -\mathrm{p} + 1, \ldots, -2, -1$

and *A* is either *Input_Noise* (if *Input_Noise* is used) or *Output_Noise* (otherwise).

Example 1

In this example, RANDOM_ARMA is used to generate a time series of length five, using an ARMA model with three autoregressive parameters and two moving average parameters. The start values are 0.1000, 0.0500, and 0.0375.

```
RANDOMOPT, set  =  123457
n  =  5
nparams  =  [3, 2]
ar  =  [0.5, 0.25, 0.125]
ma  =  [-0.5, -0.25]
r  =  RANDOM_ARMA(n, nparams, ar, ma)
PM, r, Format = "(5F10.3)",$
         Title = "                    ARMA random deviates"
                   ARMA random deviates
      0.637      0.317     -0.366     -2.122     -1.407
```

Example 2

In this example, a time series of length 5 is generated using an ARMA model with 4 autoregressive parameters and 2 moving average parameters. The start values are 0.1, 0.05 and 0.0375.

```
RANDOMOPT, set  =  123457
n  =  5
nparams  =  [3, 2]
ar  =  [0.5, 0.25, 0.125]
ma  =  [-0.5, -0.25]
wi  =  [0.1, 0.05, 0.0375]
theta0  =  1
avar  =  0.1
r  =  RANDOM_ARMA(n, nparams, ar, ma, /Accept_Reject, $
                    W_Init = wi, Const = theta0, $
                    Var_Noise = avar)
PM, r, Format = "(5F10.3)", $
```

```
               Title = "                         ARMA random deviates:"
                     ARMA random deviates:

      1.467      1.788      2.459      3.330      3.941
```

Warning Errors

`STAT_RNARM_NEG_VAR` — VAR(a) = “*Var_Noise*” = #, VAR(a) must be greater than 0. The absolute value of # is used for VAR(a).

FAURE_INIT Function

Initializes the structure used for computing a shuffled Faure sequence.

Usage

result = FAURE_INIT(*ndim*)

Input Parameters

ndim — The dimension of the hyper-rectangle.

Returned Value

A structure that contains information about the sequence.

Input Keywords

Base — The base of the Faure sequence.
Default: The smallest prime greater than or equal to *ndim*.

Skip — The number of points to be skipped at the beginning of the Faure sequence. Default:

$$\left\lfloor base^{m/2-1} \right\rfloor$$

where

$$m = \left\lfloor \log B / \log base \right\rfloor$$

and B is the largest representable integer.

Discussion

Discrepancy measures the deviation from uniformity of a point set.

The discrepancy of the point set

$$x_1, \ldots, x_n \in [0,1]^d, d \geq 1,$$

is

$$D_n^{(d)} = \sup_E \left| \frac{A(E;n)}{n} - \lambda(E) \right|,$$

where the supremum is over all subsets of $[0, 1]^d$ of the form

$$E = [0, t_1) \times \ldots \times [0, t_d), \ 0 \leq t_j \leq 1, \ 1 \leq j \leq d,$$

λ is the Lebesque measure, and

$$(E;n$$

is the number of the x_j contained in E.

The sequence x_1, x_2, … of points $[0,1]^d$ is a low-discrepancy sequence if there exists a constant $c(d)$, depending only on d, such that

$$D_n^{(d)} \leq c(d) \frac{(\log n)^d}{n}$$

for all n>1.

Generalized Faure sequences can be defined for any prime base $b \geq d$. The lowest bound for the discrepancy is obtained for the smallest prime $b \geq d$, so the keyword *Base* defaults to the smallest prime greater than or equal to the dimension.

The generalized Faure sequence x_1, x_2, …, is computed as follows:

Write the positive integer *n* in its *b-ary* expansion,

$$n = \sum_{i=0}^{\infty} a_i(n) b^i$$

where $a_i\,(n)$ are integers,

$$0 \le a_i(n) < b$$

The *j-th* coordinate of x_n is

$$x_n^{(j)} = \sum_{k=0}^{\infty}\sum_{d=0}^{\infty} c_{kd}^{(j)}\ a_d\,(n)\ b^{-k-1}, \qquad 1 \le j \le d$$

The generator matrix for the series,

$$c_{k\,d}^{(j)}$$

is defined to be

$$c_{k\,d}^{(j)} = j^{\,d-k} c_{k\,d}$$

and

$$c_{k\,d}$$

is an element of the Pascal matrix,

$$c_{k\,d} = \begin{cases} \dfrac{d!}{c!(d-c)!} & k \le d \\ 0 & k > d \end{cases}$$

It is faster to compute a shuffled Faure sequence than to compute the Faure sequence itself. It can be shown that this shuffling preserves the low-discrepancy property.

The shuffling used is the *b-ary* Gray code. The function *G(n)* maps the positive integer *n* into the integer given by its *b-ary* expansion.

The sequence computed by this function is $x(G(n))$, where x is the generalized Faure sequence.

Example

In this example, five points in the Faure sequence are computed. The points are in the three-dimensional unit cube.

Note that FAURE_INIT is used to create a structure that holds the state of the sequence. Each call to FAURE_NEXT_PT returns the next point in the sequence and updates the state structure.

```
state = FAURE_INIT(3)

p = FAURE_NEXT_PT(5, state)

PM, p
     0.333689     0.492659    0.0640654
     0.667022     0.825992     0.397399
     0.778133     0.270436     0.175177
     0.111467     0.603770     0.508510
     0.444800     0.937103     0.841843
```

FAURE_NEXT_PT Function

Computes a shuffled Faure sequence.

Usage

result = FAURE_NEXT_PT(*npts, state*)

Input Parameters

npts — The number of points to generate in the hyper-rectangle.

state — State structure created by a call to FAURE_INIT.

Returned Value

An array of size *npts* by *state.dim* containing the *npts* next points in the shuffled Faure sequence.

Input Keywords

Double **—** If present and nonzero, double precision is used.

Output Keywords

Skip **—** The current point in the sequence. The sequence can be restarted by initializing a new sequence using this value for *Skip*, and using the same dimension for *ndim*.

Discussion

Discrepancy measures the deviation from uniformity of a point set.

The discrepancy of the point set

$$x_1, \ldots, x_n \in [0,1]^d \ , d \geq 1,$$

is

$$D_n^{(d)} = \sup_E \left| \frac{A(E;n)}{n} - \lambda(E) \right|,$$

where the supremum is over all subsets of $[0, 1]^d$ of the form

$$E = [0, t_1) \times \ldots \times [0, t_d),\ 0 \leq t_j \leq 1,\ 1 \leq j \leq d,$$

λ is the Lebesque measure, and

$$(E;n$$

is the number of the x_j contained in *E*.

The sequence $x_1, x_2, \ldots$ of points $[0,1]^d$ is a low-discrepancy sequence if there exists *a* constant *c(d)*, depending only on *d*, such that

$$D_n^{(d)} \leq c(d) \frac{(\log n)^d}{n}$$

for all $n>1$.

Generalized Faure sequences can be defined for any prime base $b \geq d$. The lowest bound for the discrepancy is obtained for the smallest prime $b \geq d$, so the keyword Base defaults to the smallest prime greater than or equal to the dimension.

The generalized Faure sequence $x_1, x_2, \ldots$, is computed as follows:

Write the positive integer n in its b-*ary* expansion

$$n = \sum_{i=0}^{\infty} a_i(n) b^i$$

where $a_i(n)$ are integers,

$$0 \leq a_i(n) < b$$

The j-th coordinate of x_n is

$$x_n^{(j)} = \sum_{k=0}^{\infty} \sum_{d=0}^{\infty} c_{kd}^{(j)} \; a_d(n) \; b^{-k-1}, \qquad 1 \leq j \leq d$$

The generator matrix for the series,

$$c_{k\,d}^{(j)},$$

is defined to be

$$c_{k\,d}^{(j)} = j^{\,d-k} c_{k\,d}$$

and

$$c_{k\,d}$$

is an element of the Pascal matrix,

$$c_{k\,d} = \begin{cases} \dfrac{d!}{c!(d-c)!} & k \leq d \\ 0 & k > d \end{cases}$$

It is faster to compute a shuffled Faure sequence than to compute the Faure sequence itself. It can be shown that this shuffling preserves the low-discrepancy property.

The shuffling used is the *b-ary* Gray cod*e.* The function *G(n*) maps the positive integer *n in*to the integer given by its *b-ary* expansion.

The sequence computed by this function is *x*(*G*(*n*)), where *x* is the generalized Faure sequence.

Example

In this example, five points in the Faure sequence are computed. The points are in the three-dimensional unit cube.

Note that FAURE_INIT is used to create a structure that holds the state of the sequence. Each call to FAURE_NEXT_PT returns the next point in the sequence and updates the state structure.

```
state = FAURE_INIT(3)

p = FAURE_NEXT_PT(5, state)

PM, p

      0.333689      0.492659     0.0640654

      0.667022      0.825992      0.397399

      0.778133      0.270436      0.175177

      0.111467      0.603770      0.508510

      0.444800      0.937103      0.841843
```

3

CHAPTER

Updates to Existing Functionality

This chapter discusses the following topics:

- **Updated PV-WAVE Functions and Procedures** — Descriptions of new keywords and parameters that have been added to PV-WAVE functions and procedures. (page 239)
- **Updated PV-WAVE:IMSL Statistics Functions and Procedures** — Description of new functionality that has been added to PV-WAVE:IMSL Statistics functions and procedures. (page 241)
- **Updated PV-WAVE:IMSL Mathematics Functions and Procedures** — Description of new functionality that has been added to PV-WAVE:IMSL Mathematics functions and procedures. (page 242)

Updated PV-WAVE Functions and Procedures

This section describes new keywords and parameters that have been added to PV-WAVE functions and procedures.

JOURNAL Procedure

Keywords

Nobuffer — If present and nonzero, output lines should be written immediately to the journal file without the normal file buffering.

NoBlock Keyword for PV-WAVE VDA Tool Procedures

A new keyword, *NoBlock*, has been added for all standard VDA tool procedures (see the *PV-WAVE Reference*).

Keywords

NoBlock — If specified, the event loop (WwLoop) that is started by the VDA Tool will use the given value. By default a value of 1 (non-blocking loop) will be used. However, it may be necessary in certain circumstances to force a non-blocking loop by specifying `NoBlock =2`. (See WwLoop in the *PV-WAVE Application Developer's Guide* for more information.)

YLabelCenter Keyword

Used With Routines: AXIS, BAR, BAR2D, BAR3D, CONTOUR, CONTOUR2, OPLOT, PLOT, PLOT_FIELD, SURFACE

Corresponding System Variable: None.

Controls whether the top and bottom major tick labels on a Y axis will be positioned within the boundaries of the axis box or centered across from the corresponding major tick.

If this keyword is set, the top and bottom Y axis major tick labels will be centered vertically with corresponding major ticks. If this keyword is not set, the default behavior is to position the top and bottom Y axis major tick labels within the boundaries of the axis box.

!Version System Variable

The !Version system variable now has a field called Revision. !Version.Revision will either be a null string for major PV-WAVE releases, e.g. 7.00, 7.01, 7.10, or a letter denoting the patch revision level, e.g. 'a', 'b', etc. Printing the complete version number for PV-WAVE is now:

PRINT, !Version.release + !Version.revision

Updated PV-WAVE:IMSL Statistics Functions

This section describes new features that has been added to PV-WAVE:IMSL Statistics functions. For details, see the *PV-WAVE: IMSL Statistics Reference*.

RANDOM Function

Added several new distributions, including multinomial and stable. Also added support for random permutations and samples.

RANDOMOPT Function

Added support for generating substream seeds, and support for a Generalized Feedback Shift Register (GFSR) generator.

Updated PV-WAVE:IMSL Mathematics Functions

This section describes new features that have been added to PV-WAVE:IMSL Mathematics functions. For details, see the *PV-WAVE: IMSL Mathematics Reference*

RANDOM Function

Added several new distributions, including multinomial and stable. Also added support for random permutations and samples.

RANDOMOPT Function

Added support for generating substream seeds, and support for a Generalized Feedback Shift Register (GFSR) generator.

Index